Oliver Kullmann (Ed.)

LNCS 5584

Theory and Applications of Satisfiability Testing – SAT 2009

12th International Conference, SAT 2009
Swansea, UK, June/July 2009
Proceedings

Lecture Notes in Computer Science 5584

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Oliver Kullmann (Ed.)

Theory and Applications of Satisfiability Testing – SAT 2009

12th International Conference, SAT 2009
Swansea, UK, June 30 - July 3, 2009
Proceedings

Springer

Volume Editor

Oliver Kullmann
Computer Science Department
Swansea University
Faraday Building, Singleton Park
Swansea, SA2 8PP, UK
E-mail: o.kullmann@swansea.ac.uk

Library of Congress Control Number: Applied for

CR Subject Classification (1998): F.4.1, I.2.3, I.2.8, I.2, F.2.2, G.1.6

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-02776-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-02776-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12712779 06/3180 5 4 3 2 1 0

Preface

This volume contains the papers presented at SAT 2009: 12th International Conference on Theory and Applications of Satisfiability Testing, held from June 30 to July 3, 2009 in Swansea (UK).

The International Conference on Theory and Applications of Satisfiability Testing (SAT) started in 1996 as a series of workshops, and, in parallel with the growth of SAT, developed into the main event for SAT research. This year's conference testified to the strong interest in SAT, regarding theoretical research, research on algorithms, investigations into applications, and development of solvers and software systems. As a core problem of computer science, SAT is central for many research areas, and has deep interactions with many mathematical subjects. Major impulses for the development of SAT came from concrete practical applications as well as from fundamental theoretical research. This fruitful collaboration can be seen in virtually all papers of this volume.

There were 86 submissions (completed papers within the scope of the conference). Each submission was reviewed by at least three, and on average 4.0 Programme Committee members. The Committee decided to accept 45 papers, consisting of 34 regular and 11 short papers (restricted to 6 pages). A main novelty was a "shepherding process", where 29% of the papers were accepted only conditionally, and requirements on necessary improvements were formulated by the Programme Committee and its installment monitored by the "shepherd" for that paper (using possibly several rounds of feedback). This process helped enormously to improve the quality of the papers, and it also enabled the Programme Committee to accept 13 papers, which have very interesting contributions, but which due to weaknesses normally wouldn't have made it into the proceedings. 27 regular and 5 short papers were accepted unconditionally, and 7 long and $7 = 3 + 4$ short papers were accepted conditionally (with 4 required conversions from regular to short papers). All these 7 long papers and 6 of the 7 short papers could then be accepted in the "second round", involving in all cases substantial work for the authors (often a complete revision) and the shepherd (ranging from providing general advice to complete grammatical overhauls). As one author put it: "I would, however, like to congratulate the reviewers, as their review is the most useful and thorough I have ever received from any conference - indeed, if integrated correctly, it brings a new level of quality to the paper."

The organisation of the papers is by subjects (and within the categories alphabetically). The programme included two invited talks:

- Robert Niewenhuis considered how SMT ("SAT modulo theories") can enhance SAT solving in a systematic way by special algorithms, as it is possible in constraint programming.
- Moshe Vardi investigated how the strong inference power delivered by OBDDs ("ordered binary decision diagrams") can be harnessed by SAT solving.

One of the major topics of this conference was the MAXSAT problem (maximising the number of satisfied clauses), and boolean optimisation problems in general. Besides these extensions, the papers of this conference show that "core SAT", that is, boolean CNF-SAT solving, has still a huge potential (I expect that we just scratched the surface, and fascinating discoveries are waiting for us). One fundamental topic was the understanding of why and when SAT solvers are efficient, and interesting approaches were considered, towards a more precise intelligent control of the execution of SAT solvers. Another strong area of this year was the intelligent translation of problems into SAT. Regarding QBF, the extension of SAT by allowing quantification, the quest for a "good" problem representation becomes even more urgent, and we find theoretical and practical approaches.

Several additional events were associated with the SAT conference, including the SAT competition, the PB competition ("pseudo-boolean", allowing certain forms of arithmetic), the Max-SAT evaluation, and a special session on the various aspects of the process of developing SAT software.

Arnold Beckmann and Matthew Gwynne helped with the local organisation. We gladly acknowledge the following people in organising the satellite events:

- the main organisers of the SAT competition Daniel Le Berre, Olivier Roussel, Laurent Simon, the judges Andreas Goerdt, Inês Lynce and Aaron Stump, and the special organisers Allen Van Gelder, Armin Biere, Edmund Clarke, John Franco and Sean Weaver
- the organisers of the PB competition Vasco Manquinho and Olivier Roussel;
- and the organisers of the Max-SAT evaluation Josep Argelich, Chu Min Li, Felip Manyà and Jordi Planes

A special thanks goes to the Programme Committee and the additional external reviewers, who through their thorough and knowledgeable work enabled the assembly of this body of high-quality work. We also thank the authors for their enthusiastic collaboration in further improving their papers.

The EasyChair conference management system helped us with handling of the paper submissions, paper reviewing, paper discussion and assembly of the proceedings. I would like to thank the Chairs of the previous years, Hans Kleine Büning, Xishun Zhao and Joao Marques-Silva, for their important advice on running a conference. The Department of Computer Science of Swansea University provided logistic support. Finally I would like to thank the following sponsors for their support of SAT 2009: Intel Corporation, NEC Laboratories, and Invensys Rail Group.[1]

April 2009 Oliver Kullmann

[1] Due to the difficult economic circumstances a number of former sponsors expressed their regret for not being able to provide funding this year.

Conference Organisation

Conference and Programme Chair

Oliver Kullmann	Computer Science Department, Swansea University, UK

Local Organisation

Arnold Beckmann	Computer Science Department, Swansea University, UK
Matthew Gwynne	Computer Science Department, Swansea University, UK

Programme Committee

Dimitris Achlioptas
Armin Biere
Stephen Cook
Nadia Creignou
Evgeny Dantsin
Adnan Darwiche
John Franco
Nicola Galesi
Enrico Giunchiglia
Ziyad Hanna
Marijn Heule
Edward Hirsch
Kazuo Iwama
Hans Kleine Büning
Daniel LeBerre
Chu Min Li
Ines Lynce
Panagiotis Manolios
Joao Marques-Silva
David Mitchell
Albert Oliveras
Ramamohan Paturi
Lakhdar Sais
Karem Sakallah
Uwe Schöning
Roberto Sebastiani
Robert Sloan
Carsten Sinz
Niklas Sörensson
Ewald Speckenmeyer
Stefan Szeider
Armando Tacchella
Miroslaw Truszczynski
Alasdair Urquhart
Allen Van Gelder
Hans van Maaren
Toby Walsh
Sean Weaver
Emo Welzl
Lintao Zhang
Xishun Zhao

External Reviewers

Anbulagan Anbulagan
Carlos Ansótegui
Josep Argelich
Regis Barbanchon
Maria Luisa Bonet
Simone Bova
Roberto Bruttomesso
Uwe Bubeck
Lorenzo Carlucci
Harsh Raju Chamarthi
Benjamin Chambers
Hubie Chen
Gilles Dequen
Laure Devendeville
Juan Luis Esteban
Paulo Flores
Anders Franzen
Heidi Gebauer
Eugene Goldberg
Alexandra Goultiaeva
Alberto Griggio
Djamal Habet
Shai Haim
Miki Hermann

Dmitry Itsykson
George Katsirelos
George Katsirelose
Arist Kojevnikov
Stephan Kottler
Alexander Kulikov
Javier Larrosa
Silvio Lattanzi
Massimo Lauria
Jimmy Lee
Theodor Lettmann
Florian Lonsing
Toni Mancini
Vasco Manquinho
Felip Manyà
Marco Maratea
Paolo Marin
John Moondanos
Robin Moser
Massimo Narizzano
Nina Naroditskaya
Sergey Nikolenko
Sergey Nurk
Richard Ostrowski
Cédric Piette
Knot Pipatsrisawat
Jordi Planes
Stefan Porschen
Luca Pulina
Silvio Ranise
Andreas Razen
Alyson Reeves
Olivier Roussel
Emanuele Di Rosa
Jabbour Said
Dominik Scheder
Thomas Schiex
Tatjana Schmidt
Henning Schnoor
Yuping Shen
Michael Soltys
Stefano Tonetta
Patrick Traxler
Enrico Tronci
Gyorgy Turan
Olga Tveretina
Alexander Wolpert
Stefan Woltran
Grigory Yaroslavtsev
Weiya Yue
Bruno Zanuttini
Michele Zito
Philipp Zumstein

Sponsoring Institutions

Computer Science Department, Swansea University
Invensys Rail Group
Intel Corporation
NEC Laboratories

Table of Contents

5. Resolution and SAT

6. Translations to CNF

7. Techniques for Conflict-Driven SAT Solvers

8. Solving SAT by Local Search

9. Hybrid SAT Solvers

10. Automatic Adaption of SAT Solvers

11. Stochastic Approaches to SAT Solving

12. QBFs and Their Representations

13. Optimisation Algorithms

14. Distributed and Parallel Solving

SAT Modulo Theories: Enhancing SAT with Special-Purpose Algorithms

Robert Nieuwenhuis*

During the last decade SAT techniques have become very successful for practice, with important impact in applications such as electronic design automation. DPLL-based clause-learning SAT solvers work surprisingly well on real-world problems from *many* sources, using a *single, fully automatic, push-button* strategy. Hence, modeling and using SAT is essentially a *declarative* task. On the negative side, propositional logic is a very low level language and hence modeling and encoding *tools* are required. Also, the answer can only be "unsatisfiable" (possibly with a proof) or a model: *optimization* aspects are not as well studied.

For applications such as hard/software verification, more and more complicated and sophisticated encodings into SAT were developed for constraints such as EUF (Equality with Uninterpreted Functions, i.e., congruences), Difference Logic, or other fragments of linear arithmetic.

However, it is nowadays clear that *SAT Modulo Theories* (SMT) is frequently several orders of magnitude faster. The idea is a tight integration of two components: a *theory solver* that can handle conjunctive constraints, and a DPLL-based SAT engine that does the search without knowing the semantics of the literals. Similarly to the constraint propagators in Constraint Programming (CP), the theory solver uses efficient *specialized algorithms* for detecting additional propagations and inconsistencies.

In this talk we first give an overview of our *DPLL(T)* approach to SMT and its implementation in the Barcelogic SMT tool. Then we discuss a longer-term research project, namely the development of SMT technology for hard combinatorial (optimization) problems outside the usual verification applications. Our aim is to obtain the best of several worlds, combining the advantages inherited from SAT: efficiency, robustness and automation (no need for *tuning*) and CP features such as rich modeling languages, special-purpose filtering algorithms (for, e.g., planning, scheduling or timetabling constraints), and sophisticated optimization techniques. We give several examples and discuss the impact of aspects such as first-fail heuristics vs activity-based ones, realistic structured problems vs random or handcrafted ones, and lemma *learning*.

* Technical Univ. of Catalonia (UPC), Barcelona, Spain. Partially supported by Spanish Min. of Science &Innovation, LogicTools-2 project (TIN2007-68093-C02-01). For more details and further references, see Robert Nieuwenhuis, Albert Oliveras and Cesare Tinelli: Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T), *Journal of the ACM*, 53(6), 937-977, 2006.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, p. 1, 2009.

Symbolic Techniques in Propositional Satisfiability Solving*

Moshe Y. Vardi

Rice University, Department of Computer Science, Houston, TX 77251-1892, U.S.A.
vardi@cs.rice.edu
http://www.cs.rice.edu/~vardi

Search-based techniques in propositional satisfiability (SAT) solving have been enormously successful, leading to what is becoming known as the "SAT Revolution". Essentially all state-of-the-art SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) technique, augmented with backjumping and conflict learning. Much of current research in this area involves refinements and extensions of the DPLL technique. Yet, due to the impressive success of DPLL, little effort has gone into investigating alternative techniques. This work focuses on symbolic techniques for SAT solving, with the aim of stimulating a broader research agenda in this area.

Refutation proofs can be viewed as a special case of constraint propagation, which is a fundamental technique in solving constraint-satisfaction problems. The generalization lifts, in a uniform way, the concept of refutation from Boolean satisfiability problems to general constraint-satisfaction problems. On the one hand, this enables us to study and characterize basic concepts, such as refutation width, using tools from finite-model theory. On the other hand, this enables us to introduce new proof systems, based on representation classes, that have not been considered up to this point. We consider ordered binary decision diagrams (OBDDs) as a case study of a representation class for refutations, and compare their strength to well-known proof systems, such as resolution, the Gaussian calculus, cutting planes, and Frege systems of bounded alternation-depth. In particular, we show that refutations by ODBBs polynomially simulate resolution and can be exponentially stronger.

We then describe an effort to turn OBDD refutations into OBBD decision procedures. The idea of this approach, which we call *symbolic quantifier elimination*, is to view an instance of propositional satisfiability as an existentially quantified propositional formula. Satisfiability solving then amounts to quantifier elimination; once all quantifiers have been eliminated we are left with either **1** or **0**. Our goal here is to study the effectiveness of symbolic quantifier elimination as an approach to satisfiability solving. To that end, we conduct a direct comparison with the DPLL-based ZChaff, as well as evaluate a variety of optimization techniques for the symbolic approach. In comparing the symbolic approach to ZChaff, we evaluate scalability across a variety of classes of formulas. We find that no approach dominates across all classes. While ZChaff dominates for many classes of formulas, the symbolic approach is superior for other classes of formulas.

* Work supported in part by NSF grants CCR-0311326, CCF-0613889, ANI-0216467, and CCF-0728882.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 2–3, 2009.

Finally, we turn our attention to Quantified Boolean Formulas (QBF) solving. Much recent work has gone into adapting techniques that were originally developed for SAT solving to QBF solving. In particular, QBF solvers are often based on SAT solvers. Most competitive QBF solvers are search-based. Here we describe an alternative approach to QBF solving, based on symbolic quantifier elimination. We extend some symbolic approaches for SAT solving to symbolic QBF solving, using various decision-diagram formalisms such as OBDDs and ZDDs. In both approaches, QBF formulas are solved by eliminating all their quantifiers. Our first solver, QMRES, maintains a set of clauses represented by a ZDD and eliminates quantifiers via multi-resolution. Our second solver, QBDD, maintains a set of OBDDs, and eliminate quantifiers by applying them to the underlying OBDDs. We compare our symbolic solvers to several competitive search-based solvers. We show that QBDD is not competitive, but QMRESS compares favorably with search-based solvers on various benchmarks consisting of non-random formulas.

References

1. Atserias, A., Kolaitis, P.G., Vardi, M.Y.: Constraint propagation as a proof system. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 77–91. Springer, Heidelberg (2004)
2. Pan, G., Vardi, M.Y.: Symbolic decision procedures for QBF. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 453–467. Springer, Heidelberg (2004)
3. Pan, G., Vardi, M.Y.: Search vs. symbolic techniques in satisfiability solving. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 235–250. Springer, Heidelberg (2005)
4. Pan, G., Vardi, M.Y.: Symbolic techniques in satisfiability solving. J. of Automated Reasoning 35, 25–50 (2005)

Efficiently Calculating Evolutionary Tree Measures Using SAT

Maria Luisa Bonet[1] and Katherine St. John[2]

[1] Lenguajes y Sistemas Informáticos, Universidad Politécnica de Cataluña, Spain
[2] Math & Computer Science Dept., Lehman College, City U. New York, USA

Abstract. We develop techniques to calculate important measures in evolutionary biology by encoding to CNF formulas and using powerful SAT solvers. Comparing evolutionary trees is a necessary step in tree reconstruction algorithms, locating recombination and lateral gene transfer, and in analyzing and visualizing sets of trees. We focus on two popular comparison measures for trees: the hybridization number and the rooted subtree-prune-and-regraft (rSPR) distance. Both have recently been shown to be NP-hard, and efficient algorithms are needed to compute and approximate these measures. We encode these as a Boolean formula such that two trees have hybridization number k (or rSPR distance k) if and only if the corresponding formula is satisfiable. We use state-of-the-art SAT solvers to determine if the formula encoding the measure has a satisfying assignment. Our encoding also provides a rich source of real-world SAT instances, and we include a comparison of several recent solvers (minisat, adaptg2wsat, novelty+p, Walksat, March KS and SATzilla).

1 Introduction

Phylogenies, or evolutionary histories, play a central role in biology. While traditionally represented as trees, due to evolutionary processes such as hybridization, horizontal gene transfer and recombination [16], the relationship between many species is better represented by networks, or directed graphs. These nontree events connect nodes from different branches of a tree, and they are usually called *reticulations* (see Figure 1). Given two trees that represent the evolutionary history of different genes of a set of species, the *hybridization number* between the trees characterizes the number of reticulation events needed to explain the evolution of the set of species. With the recent explosion in biological data available, it is now possible to compute multiple phylogenetic trees for a set of taxa (species), based on many different gene sequences. Calculating the differences between species and gene trees very efficiently is essential to building evolutionary histories, and in turn to understanding the underlying properties of the species. Further, comparing phylogenies play important roles in locating recombination and lateral gene transfers, and analyzing searches in treespace.

Our primary focus is on calculating the hybridization number. The related rooted subtree-prune-and-reconnect (rSPR) distance is often used as a surrogate.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 4–17, 2009.

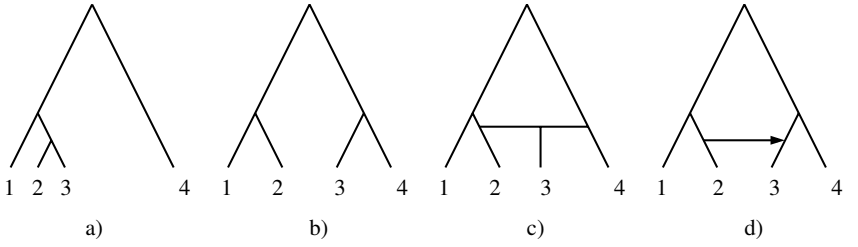


Fig. 1. Hybridization events: a) and b) represent two different gene trees on the same set of species, and c) and d) show two possible evolutionary scenarios. In c), species 2 and 4 hybridize (combine genetic information) to form a new species 3. In d), we show lateral gene transfer where some of the genetic information from species 3 is derived along one lineage as in tree in a), while other information is derived along the lineages shown in b).

rSPR captures individual hybridization events but misses an important acyclicity condition that taxa cannot have themselves as ancestors. Further, while often similar in size, there exist instances where the difference between the rSPR and hybridization number are arbitrarily large [5].

Calculcating tree measures is of great interest, and the focus of much recent work. Bordewich and Semple [6] showed that the hybridization number is NP-hard and fixed parameter tractable, by relating it with an appropriately defined agreement forest. Agreement forests were developed for evolutionary tree metrics in the pioneering work of Hein *et al.* [14] and Allen and Steel [1] that linked the tree distance to the size of the maximum agreement forest (MAF). With the development of a MAF for the rooted subtree-prune-and-reconnect (rSPR) distance [5] (see Figure 2), Bonet *et al.* [4] showed these algorithms are a 5-approximation for rSPR distance. Algorithms for biologically relevant restricted cases of rSPR were also developed by Hallett and Lagergren [13] and Beiko and Hamilton [3]. Nakhleh *et al.* [20] developed a very fast heuristic for rSPR distance, which due to its basis on maximum agreement subtrees, also yields bounds on the hybridization number. Wu [28] encodes the rSPR problem into an integer linear programming instance, achieving good results for the rSPR problem only. To find exact answers for hybridization numbers, Linz *et al.* [7] used clever combinatorial characterizations to yield an exhaustive search that does well for surprisingly large values.

We have developed new software tools to calculate hybridization number and rSPR distance, by transforming these into satisfiability (SAT) questions. Using combinatorial characterizations and insights of past work, we can often reduce the scope of the problem to several smaller subproblems for hybridization, or a single smaller problem for rSPR. We use two different approaches to calculating these measures: exact calculation and an upper bound heuristic. Our novel contribution is the use of powerful SAT solvers to finish this final part of the computation on the reduced trees. We do this by encoding the problem as a Boolean formula such that two trees have some particular or hybrid number

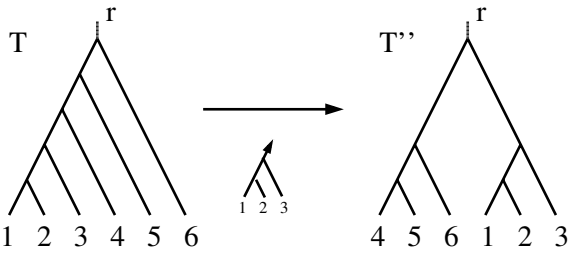


Fig. 2. rSPR Move: A rooted SPR move breaks off a subtree from the first tree and reattaches the subtree to another tree. For technical reasons, we represent our rooted trees as "planted trees" and allow rSPR moves to reattach subtree to the edge of the root, as done with the rSPR move above.

(or rSPR distance) if and only if the corresponding formula is satisfiable. Then we give the formula as input to one of the best SAT solvers. Due to the large community focused on techniques to solve SAT more efficiently, there are many different choices of SAT solvers, optimized for differing criteria.

For our upper bound heuristic (SAT Descent), we work down from an upper bound (instead of eliminating possibilities counting up from zero). In this case we do a comparison among several solvers. They are walksat [24,25], adaptg2wsat [8], novelty+p [8], minisat [10,11], SATzilla [29] and March KS [15]. Notice that we compare all kinds of different solvers: local search algorithms (the first three), DPLL with learning (minisat), SAT solver portfolio (SATzilla) and solver specialized on random instances (March KS). The performance of minisat on our instances was worse in general than the performance of the local search solvers. Using local search algorithms yields excellent results in both accuracy and performance. For example, we find solutions for biological data sets in 48 seconds that take over 11 hours with the exact program, HybridNumber and do not finish after two days of compute time using the complete solver minisat.

This paper is organized as follows: we give background on tree measures and agreement forests in Section 2. Section 3 details our methods, with more information on the SAT encoding in Section 4. Section 5 describes the data analyzed. Results are in Section 6, followed by discussion and future work in Section 7.

2 Hybridization Networks and Agreement Forests

The recent theoretical results have linked tree measures to the size of maximum agreement forests [14]. This link has been used to show NP-hardness, fixed parameter tractability, and is the basis for approximation algorithms. Roughly, each measure corresponds to the size of the appropriately defined maximum agreement forest. For a more thorough treatment, see [5,18,26].

Subtree Prune and Regraft (SPR). A **subtree prune and regraft** (SPR) operation [1] on a binary tree T is defined as cutting any edge and thereby

pruning a subtree t, then regrafting the subtree by the same cut edge to a new vertex obtained by subdividing a pre-existing edge in $T - t$. We apply a forced contraction to maintain the binary property of the resulting tree (see Figure 2). The **SPR distance** between two trees T_1 and T_2 is the minimal number of SPR moves needed to transform T_1 into T_2. When working with rooted trees, we refer to this distance as **rooted SPR** or **rSPR**. Bordewich and Semple [5] showed that the rSPR distance of two trees is the same as the size of an appropriately defined maximum agreement forest for rooted trees of the two trees. This number is related to another measure between trees that we next define.

Hybridization Number. A **hybridization network** on a leaf set X [5,26] is a rooted acyclic directed graph with root ρ in which

- X is the set of leaves (vertices of outdegree zero);
- $d^+(\rho) \geq 2$;
- for all the vertices v with $d^+(v) = 1$, we have $d^-(v) \geq 2$.

Let $d^-(v)$ be the indegree of v and $d^+(v)$ be the outdegree of v. The vertices with indegree at least two represent the hybridization vertices. Now, we define the **hybridization number** of a hybridization network H with root ρ as

$$h(H) = \sum_{v \neq \rho} (d^-(v) - 1).$$

Let T be a rooted phylogenetic tree and H a hybridization network. We say H **displays** T [5,26] if T can be obtained from H by first deleting a subset of edges of H and any resulting isolated vertices, and then contracting edges. Then given two trees T_1 and T_2,

$$h(T_1, T_2) = min\{h(H) : H \text{ is a hybridization network that displays } T_1 \text{and } T_2\}.$$

We define the **hybridization number** of two trees T_1 and T_2 as the minimal hybridization number of all hybridization network H that display T_1 and T_2.

Agreement Forest. Originally linked to tree measures [14], agreement forests are an essential tool for calculating and showing hardness for tree measures. Roughly, an **agreement forest** for T_1 and T_2 with identical leaf set X, is a set of subtrees that occur in both the initial trees T_1 and T_2, where:

1. The subtrees partition the leaf set X into $\{X_0, \ldots, X_k\}$.
2. The subtrees occur as induced subtrees of T_1 and T_2. i.e. for each i, $0 \leq i \leq k$, T_1 restricted to the set of leaves X_i, and T_2 restricted to the set of leaves X_i are the ith subtree.
3. The subtrees are vertex disjoint in both T_1 and T_2.

For two trees, T_1 and T_2, with the same leaf set, a **maximum agreement forest (MAF)** is an agreement forest with the minimal number of subtrees. Allen and Steel [1] show the size of the MAF corresponds to another tree measure, the tree-branch-and-reconnect (TBR) distance. Augmenting this forest definition to handle rooted trees, Bordewich and Semple [5] link these new MAFs to rSPR distance. Figure 3 illustrates agreement forests for rSPR distance.

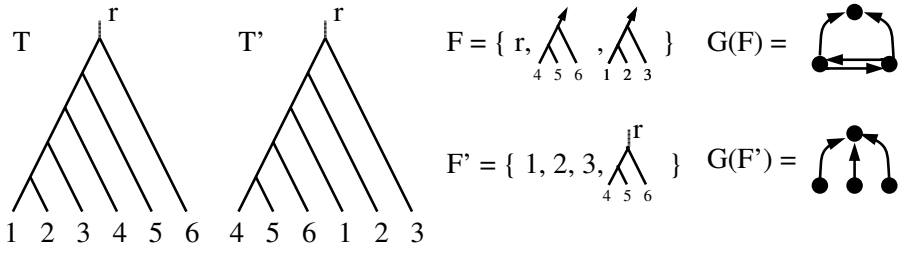


Fig. 3. Agreement Forests: F and F' are two possible forests for the trees T and T'. F is also maximal for rSPR, but its associated graph, $G(F)$ contains a cycle and is thus not a good agreement forest for hybridization. The second, larger forest, is acyclic, and is the maximum agreement forest for hybridization. The rSPR distance is 2, while the hybrid number is 3.

Hybrid Number and Acyclicity of the Forest. We define the graph, G_F of a MAF F of two trees T_1 and T_2 as follows: the nodes are the trees of F, and there is an edge from one node (F_1) to (F_2) corresponding to two trees of F if the root of (F_1) is a descendant of the root of (F_2) in either T_1 or T_2. Adding the simple condition that the graph of the forest is acyclic yields a MAF for hybridization number. That is, a forest that is maximal with respect to all agreement forests that have acyclic associated graphs has size equivalent to the hybridization number of the two trees [6]. See Figure 3.

Hardness Results. Both of these measures, hybridization number and rSPR distance have been shown to be NP-hard and fixed parameter tractable [5,6]. The following operations help reduce the size of the trees and provide additional efficiency for our methods by "shrinking" the size of the problem encoded:

Subtree Reduction (Rule 1 of [5]). Replace any pendant subtree that occurs identically in both trees T_1 and T_2 by a single leaf with a new label.

Our second rule looks at clusters in trees. While not part of the fixed parameter tractability reduction for hybridization number, it gives important reductions on the sizes of the trees and improves the performance. A is a **cluster** for T_1 and T_2 if there is a node in each tree that has A as its set of descendants in X. We note that this reduction preserves hybridization number but does not preserve rSPR distance [2]:

Cluster Reduction (Rule 3 of [2]). Let T_1 and T_2 be two rooted binary X-trees, and $A \subset X$ a cluster of both T_1 and T_2. Then,

$$h(T_1, T_2) = h(T_1 \mid A, T_2 \mid A) + h(T_{1a}, T_{2a})$$

where T_{1a} (T_{2a}) is the result of substituting the subtree of T_1 (T_2) having leaf set A by the new leaf a and $T_1 \mid A$ ($T_2 \mid A$) is the restriction of T_1 (T_2) to A.

3 Methods

We develop four related algorithms for calculating the tree measures: exact solutions ('SAT Ascent') and upper bound heuristics ('SAT Descent') for both hybridization number and rSPR distance. Our input is two trees, T_1 and T_2, that represent the evolution of two different genes of a set of species. Our methods break into several parts:

1. Efficient preprocessing to reduce size, using known reductions (see §2),
2. Encoding the questions "hybridNumber$(T_1, T_2) = r$?" and "$d_{rSPR}(T_1, T_2) = r$?" as Boolean formulas,
3. Using fast heuristics [20] to give starting upper bounds, and
4. Using different search strategies and solvers to answer these questions.

Efficient Preprocessing. Each of the reduction rules can be performed in linear time, following a clever coding of trees by Day [9]. His coding stores sufficient information about each internal vertex to identify internal structure. This takes $O(1)$ space per internal vertex, allowing linear time algorithms for the reduction rules presented in the previous section (see [4] for more details).

Encoding. We describe the SAT encoding in more detail in the next section.

Efficient Heuristics. We use RIATA-HGT from the PhyloNet program suite [20] to give starting points for our upper bounds. While not an approximation algorithm (since families of trees can be constructed whose distance is fixed, but whose distance found by the algorithm is arbitrarily large), RIATA-HGT performs very well in practice (see Figures 4 and 5). It takes the input trees and calculates a maximum agreement subtree. The maximum agreement subtree is added to the forest and then used as a "backbone" and the algorithm is then repeated for each subtree hanging from the backbone. While not explicitly stated, the resulting forest is acyclic by construction and thus gives an upper bound for both rSPR distance and hybridization number.

Different Search Strategies and SAT Solvers. We use Minisat [10,11] to find exact solutions for rSPR and hybrid number. On the other hand, we use Walksat [24,25], adaptg2wsat [8], novelty+p [8] for the upper bounds of both measures. We use the UBCSAT implementation [27] for the latter two since it was significantly faster than the stand-alone versions. We compare the performance of these three local search solvers among themselves and also with the performance of the complete solvers minisat,March KS and SATzilla. As we will see in the experimentation, the local search algorithms work much faster in general.

Software. We built four different methods that calculate upper bounds for hybridization numbers, upper bounds for d_{rSPR}, exact solutions for hybridization number, and exact solutions for d_{rSPR}. The software is written in perl and java, using the TreeJuxtaposer [19] java code base. All four have similar format, so, we only describe the upper bound for hybridization numbers in detail:

trees [23]	# of taxa	Hybrid Number [7]	SAT Exact	RIATA -HGT [20]	SAT Descent w [24]	a [8]	n [8]	m [11]	z [29]
ndhf	40	14	≥ 9	15	14	16	14	≤ 15	16
phyB		11h	2d	11s	4m	24s	48s	6h	44s
ndhf	36	13	≥ 9	16	13	17	14	≤ 14	18
rbcl		11.8h	2d	11s	4m	28s	51s	6h	48s
ndhf	34	12	≥ 9	15	12	15	12	≤ 12	15
rpoC2		26.3 h	2d	7s	3m	14s	35s	6h	34s
ndhf	19	9	9	9	9	10	9	≤ 9	10
waxy		5m	46h	3s	19s	4s	7s	6h	2m
ndhf	46	≥ 15	≥ 9	24	22	22	21	≤ 20	22
xits		2d	2d	12s	3m	50s	1.2m	6h	1m
phyB	21	4	4	4	4	4	5	4	4
rbcl		1s	6s	4s	7s	4s	4s	3s	5s
phyB	21	7	7	7	7	7	7	7	10
rpoC2		3m	1.5m	3s	33s	11s	13s	77s	11s
phyB	14	3	3	3	3	3	4	3	3
waxy		1s	3s	2s	5s	3s	2s	2s	2s
phyB	30	8	8	9	8	9	9	8	10
xits		19s	1.5h	6s	1m	10s	11s	1.7h	10s
rbcl	26	13	9	16	14	15	15	≤ 15	14
rpoC2		29.5h	2d	5s	1m	9s	10s	6h	36s
rbcl	12	7	7	7	7	7	7	7	8
waxy		4m	42s	1s	10s	3s	3s	40s	7s
rbcl	29	≥ 9	≥ 9	15	14	19	14	≤ 15	19
xits		2d	2d	6s	271s	20s	1m	6h	40s
rpoC2	10	1	1	1	1	1	1	1	1
waxy		1 s	1s	1s	3s	1s	1s	1s	1s
rpoC2 xits	31	≥ 10	≥ 9	17	15	18	15	≤ 15	18
		2d	2d	7s	4m	18s	50s	6h	1h
waxy	15	8	8	10	9	10	9	8	9
xits		10m	1s	2s	13s	6s	11s	1m	14s

Fig. 4. The Grass (Poaceae) Data Set: We compare the exact solver, HybridNumber [7], the fast heuristic, RIATA-HGT [20], and our program using the SAT encodings. The data for HybridNumber in the third column is from [7]. First: HybridNumber finds the exact solution, but due to the NP-hardness of the problem, often does not find a solution. Second: The performance of the SAT Ascent solver which works upward from the smallest distance until the true distance is found. Its performance echos HybridNumber. Third: RIATA-HGT gives very quickly a reasonable, but not tight, upper bound. Right: Our software gives excellent results in reasonable time. It employs five different solvers: the incomplete solvers: Walksat [24,25] and two high scoring solvers from SAT 2007: adaptg2wsat and novelty+p [8] implemented in [27], as well as the complete solvers minisat [11] and SATzilla [29]. Solutions listed as upper or lower bounds did not halt before the time limit and estimates based on the log files are listed.

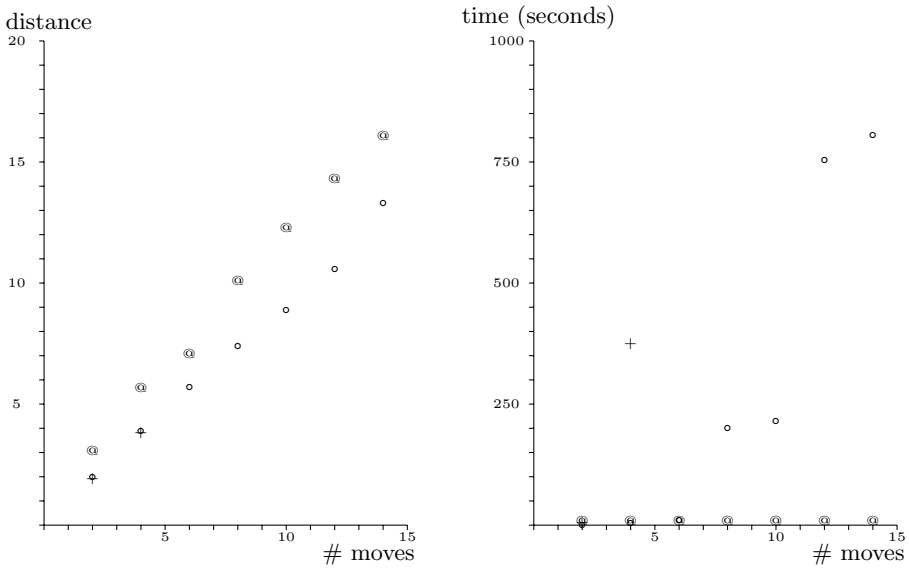


Fig. 5. Simulated Data Set: 50-taxa trees were generated under the Yule-Harding distribution to be the "species tree" and then for each distance and each species tree, 10 "gene trees" of that distance were generated. In both graphs, @ is RIATA-HGT [20], ∘ is the SAT Descent using Walksat [25], and + is the exact algorithm HybridNumber [7]. Due to the similarity in results to HybridNumber, the results for SAT Ascent solution are omitted. All runs had a 24 hour time limit. This did not affect RIATA-HGT and SAT Descent, but limited the runs that completed for HybridNumber to values 2 and 4. The left graph shows the hybridization number returned by the programs; the right graph shows the time, in seconds, to accomplish the task.

1. Preprocess by the reduction rules to yield smaller pairs of trees.
2. Find a starting upper bound for each pair using RIATA-HGT [20].
3. Starting with the upper bound, r, encode the formula for hybridization is r and use a SAT solver to find a satisfiable assignment (i.e. a MAF).
4. Decrement r and loop to 3, until a satisfiable assignment is *not* found. Return $r + 1$.

We similarly define the algorithm for upper bounds for d_{rSPR}. For the SAT Ascent algorithm, we begin by looking for an agreement forest of size 1 and work upwards until a forest is found.

4 Encoding

Our program takes pairs of phylogenetic trees on the same leaf set and a proposed size for the MAF and produces SAT instances in DIMACS SAT format:

Input: Two trees, T_1 and T_2, and an integer $r > 0$.
Output: An encoding into a SAT instance, in the DIMACS SAT format.

The resulting formula will be satisfiable if the hybridization number (rSPR distance) between T_1 and T_2 is $\leq r$. We rely on the correspondence to agreement forests, described in Section 2. Namely, that $d_{rSPR}(T_1, T_2) = r$ iff there is a maximum agreement forest for T_1 and T_2 of size r. Similarly, the hybridization number of T_1 and T_2 is r iff there is a maximum *acyclic* agreement forest for T_1 and T_2 of size r. Thus, most of the encoding focuses on saying that a agreement forest exists:

Literals. For each subtree i in the forest and leaf j from the original leaf set, we have a literal l_{ij} which is true iff leaf j is part of subtree i in the agreement forest. We have similar sets of literals for internal vertices of T_1 and T_2. We also have literals to reduce the number of clauses needed (explained below) and to represent the acyclic conditions. The number of literals is $O(rn + r^2)$. Since $r < n$, this yields $O(nr)$.

Clauses for Subtrees Partition Leaf Sets. It is easy to say that every leaf is in at least one subtree, by having clauses for each leaf j, $l_{0j} \vee l_{1j} \vee \ldots \vee l_{rj}$, that literally say, "leaf j is in subtree 0 or leaf j is in subtree 1 or ... leaf j is in subtree r. This takes $O(rn)$ clauses.

To say that every leaf occurs in at most one subtree is more difficult. The obvious encoding takes $O(rn^2)$. Following [17], we introduce $O(rn)$ new literals, s_{ij} and use them to reduce the number of clauses needed to $O(rn)$. The intuition for these new literals and corresponding clauses is that they encode $\sum_i l_{ij} \leq 1$. The new variables signal when leaf j occurs in some tree i, and the clauses ensure that this happens for only one i.

Clauses for Subtrees Occurring as Induced Trees. The clauses below assert that the $r + 1$ subtrees occur in both T_1 and T_2. This is done in a similar manner as above: we show that every internal vertex is in at most one subtree. Note that we do not need to say that every internal node is in at least one subtree. We need new variables to say to which subtrees of the agreement forest the internal vertices of T_1 and of T_2 belong to. If a rooted binary tree has n leaves, then it has $n - 1$ internal vertices. For tree T_1, we have variables v_{ij}, for $0 \leq i \leq r$ and $1 \leq j \leq n - 1$ such that v_{ij} is true iff the jth internal vertex is part of the ith subtree. Similarly, for tree T_2, we have variables v'_{ij}.

We will further have two sets of variables to reduce the number of clauses needed: $t_{i,j}$ and $t'_{i,j}$ for $i = 0, \ldots, r$ and $j = 1, \ldots, n-1$ (these are similar to the s variables used for the leaves of the trees). The clauses for the internal nodes of the trees state:

1. Every internal vertex of T_1 (and of T_2) is in at most one subtree.
   This follows the same idea as in the previous step with v and t for T_1 and with v' and t' for T_2. This is done twice to require that all the internal vertices of both the input trees occur at most once in the subtrees of the forest.
2. If two leaves occur in a subtree, then internal vertices on the path between them must also occur in the same subtree.

First, look at tree T_1 (the clauses for T_2 will be almost identical). For every pair of leaves, j and k in T_1, there exists a unique path between them of internal vertices, $v_{p_1}, v_{p_2}, \ldots, v_{p_x}$ (x and the internal vertices on the path depend on the leaves chosen and could be 0, if $i = j$, or up to $n-1$). Our clauses state that if j and k occur in subtree i, then so do the nodes on the path between them: $v_{p_1}, v_{p_2}, \ldots, v_{p_x}$. So for $i = 0, \ldots, r$ and $j, k = 1, \ldots, n-1$ we need the clauses saying

$$(l_{ij} \wedge l_{ik}) \rightarrow (v_{ip_1} \wedge v_{ip_2} \wedge \ldots \wedge v_{ip_x})$$

Note that the internal vertices and the paths *depend* on the particular tree.

Clauses for Checking that Subtrees are Equal. Once we have that the leaves form subtrees, we add clauses to guarantee that the structure of the subtrees is the same in both T_1 and T_2. This is the last condition needed to have that the subtrees form an rSPR agreement forest for T_1 and T_2. To do this, we look at triples of all leaves and their structure in T_1 and T_2. If the structure differs, then we add clauses preventing that triple of leaves from occurring in the same tree. In the worst case, this takes $O(rn^3)$ clauses, but in practice it is significantly smaller.

Clauses for Acyclic Conditions. For hybridization, the agreement forest also needs to be acyclic. Adding variables to represent that there is a directed edge between subtrees is $O(r^2)$. The clauses needed to encode the initial edges, transitive closure of the edge relationship, and forbid cycles takes $O(r^3)$.

Expected Number of Clauses. The theoretical bound on the number of clauses in this encoding is quite high, $O(rn^3)$ where n is the number of taxa in the trees and r is the hybridization number (rSPR distance) that is encoded. However, in practice, we see significantly smaller number of clauses generated by the encoding. This large difference in sizes is due to the clauses needed to check that the internal substructure of the subtrees are equal. It is possible that all the $O(n^3)$ triplets of taxa will differ in structure in T_1 and T_2, resulting in $O(rn^3)$ clauses. In practice, most trees compared have are similar and as such most of triplets agree, and few are needed. For example, the theoretical upper bound for unreduced trees with 50 taxa and with a starting upper bound of 13 is 1,625,000. For a pair chosen at random from our simulated dataset, the reduction rules shrunk the size of the trees to 39 taxa from the initial 50 taxa and the starting upper bound is 13. The number of literals and clauses depend on the size of the reduced tree pairs and the starting upper bound. They are 3,416 literals and 370,571 clauses, a huge reduction from the worst case bound for the full trees and half of the bound calculated for the reduced trees.

5 Data

We analyze both biological and simulated data. The biological data set, from the analysis of HybridNumber [7] and described more fully there, is from the

Poaceae (Grass) family. Hybridization is a well-recognized occurrence in grasses [12], making this an excellent test data set. The data set consists of sequence data for six loci: internal transcribed spacer of ribosomal DNA (ITS); NADH dehydrogenase, subunit F (ndhF); phytochrome B (phyB); ribulose 1,5-biphosphate carboxylase/oxygenase, large subunit (rbcL); RNA polymerase II, subunit (rpoC2); and granule bound starch synthase I (waxy). For each loci, a tree was built using the fastDNAmL program [21] by Heiko Schmidt [23]. As in [7], we looked at pairs of trees, reduced to their common taxa. In all, we have 15 pairs of trees. The pairs and the number of overlapping taxa are listed in Figure 4.

The simulated datasets were generated to capture small and medium distances between reasonably sized trees. All trees have 50 taxa. For each run, we generated a "species" tree, and then 10 "gene" trees by making k rSPR-moves from the species tree for $k = 2, 4, 6, 8, 10, 12, 14$. These give tree pairs with rSPR distance at most k, since it is possible for some of the sequence of moves to "cancel" each other out. The hybridization number could be larger than k, since its corresponding maximum agreement forest is that for rSPR with additional acyclic conditions. Each of the species trees was generated with Sanderson's `r8s` program [22], using Yule-Harding distribution. The program that alters the species tree by k rSPR moves chooses a non-pendant edge uniformly and at random (software written by the authors in Java). For each k, 10 trials were generated, yielding 100 species-gene tree pairs, for a total of 700 pairs of trees.

6 Results

We show the results for the hybridization number algorithms. The rSPR distance results have similar, and often worst running times, since cluster reduction rule does not apply to rSPR distance. This rule often breaks the problem into reasonably sized subproblems, speeding computation.

Poaceae (Grass) Dataset. The results for this dataset are presented in Figure 4. Our exact solution algorithm does well at small cases, as HybridNumber does but slows down for larger instances sooner. On the other hand, our SAT Descent algorithm performs extremely well using the local search algorithm, Walksat, finding the true number in 11 out of 12 of the known cases and doing so in under five minutes time. Surprisingly, Walksat outperforms more recent local search algorithms including adaptg2wsat (which recently won a silver medal in SAT2007 competition in satisfiable random formula category). All the local search algorithm outperformed the complete solvers, which often ran out of time before completing the calculations. In Figure 4, we do not include the results for March KS, since this solver performed very poorly on almost all these instances. RIATA-HGT returns answers extremely quickly, all in less than 12 seconds, but overestimates by average of 9%.

Simulated 50 Taxa Dataset. Figure 5 contains the graphs for the simulated data for both accuracy and speed. Both HybridNumber and SAT Ascent solver

could not calculate the solutions for $r \geq 6$ in the 24 hour time-limit used for these experiments. Since the SAT Ascent solver's results mirror HybridNumber, we report only the latter. Our upper bound software did extremely well in both accuracy and speed. By construction, SAT Descent with local search algorithms always gave answers that were closer to the true answer. RIATA-HGT finished in under 15 seconds for all runs. SAT Descent with local search algorithms completed all runs in less than 15 minutes. The standard deviations were omitted from Figure 5 but are worth noting. For small values of k, they are below 5% for the time and accuracy of both RIATA-HGT and SAT upper bound. The standard deviation for the time for RIATA-HGT remains below 2% for all values. For all other algorithms, the standard deviations rise for both time and accuracy to almost 20%, illustrating the variability of difficulty of problems even for small and medium values.

7 Discussion and Conclusion

Encoding problems as SAT instances has positive and negative points. On the negative side, we must build a SAT instance that may be even bigger than the original problem. On the positive side, once the hard work of encoding is done, we can use the variety of SAT tools to try many different search strategies to improve our algorithms in both efficiency and time. In a way, it is like having several solvers in one, since we can benefit from all the different tools that the SAT community has developed over the years and from future improvements of SAT solvers.

Our novel approach of encoding the NP-hard problems of calculating hybridization number and rSPR distance into SAT instances yields an elegant and efficient algorithm for estimating these measures. While not an exact answer, our algorithms often find the true answer in a fraction of the time needed to search for the exact solution. Given the ever-improving state of SAT-solvers, these results will only improve, allowing for better bounds. Future work includes improving the encoding, finding tighter bounds via combinatorial analysis of the inputs, and uses for related tree problems such as TBR distance.

One final observation is that our grass instances are an unusual case of combinatorial real problems better solved by local search algorithms than by DPLL solvers. Even though the instances come from real data, we are encoding an NP-hard problem of complexity similar to random instances, and local search solvers win the Random Satisfiable category in competitions.

Acknowledgements

The first author was partially supported by the Spanish grant TIN2007-68005-C04-03. Also, this project was partially supported by USA NSF grants SEI 0513660, and by MRI 0215942 and the computational research center at the CUNY Graduate Center. The second author would like to thank the Centre de Recerca Matemàtica in Barcelona for hosting her visits in 2007. We also thank

Charles Semple, Simone Linz, and Carlos Ansotegui for helpful conversations and the Munzner group (UBC) for the TreeJuxtaposer [19] code base.

References

1. Allen, B., Steel, M.: Subtree transfer operations and their induced metrics on evolutionary trees. Annals of Combinatorics 5, 1–13 (2001)
2. Baroni, M., Semple, C., Steel, M.: Hybrids in real time. Systematic Biology 55, 46–56 (2006)
3. Beiko, R.G., Hamilton, N.: Phylogenetic identification of lateral genetic transfer events. BMC Evol. Biol. 6, 15 (2006)
4. Bonet, M.L., John, K.S., Mahindru, R., Amenta, N.: Approximating subtree distances between phylogenies. Journal of Computational Biology 13(8), 1419–1434 (2006)
5. Bordewich, M., Semple, C.: On the computational complexity of the rooted subtree prune and regraft distance. Annals of Combinatorics 8, 409–423 (2005)
6. Bordewich, M., Semple, C.: Computing the minimum number of hybridization events for a consistent evolutionary history. Discrete Applied Mathematics (2007)
7. Bordewich, M., Linz, S., John, K.S., Semple, C.: A reduction algorithm for computing the hybridization number of two trees. Evolutionary Bioinformatics 3, 86–98 (2007)
8. Zhang, H., Li, C.M., Wei, W.: Combining adaptive noise and look-ahead in local search for SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 121–133. Springer, Heidelberg (2007)
9. Day, W.H.E.: Optimal algorithms for comparing trees with labeled leaves. Journal of Classification 2, 7–28 (1985)
10. Eén, N., Sörensson, N.: Software, http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/
11. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
12. Grass Phylogeny Working Group. Phylogeny and subfamilial classification of the grasses (poaceae). Annals of the Missouri Botanical Garden 88(3), 373–457 (2001)
13. Hallett, M.T., Lagergren, J.: Efficient algorithms for lateral gene transfer problems. In: ACM (ed.) Proceedings of the Fifth Annual International Conference on Computational Molecular Biology (RECOMB 2001), pp. 149–156. ACM, New York (2001)
14. Hein, J., Jiang, T., Wang, L., Zhang, K.: On the complexity of comparing evolutionary trees. Discrete Applied Mathematics 71, 153–169 (1996)
15. Heule, M.J.H., van Maaren, H.: March dl: Adding adaptive heuristics and a new branching strategy. Journal on Satisfiability, Boolean Modeling and Computation 2, 47–59 (2006)
16. Huson, D.H., Bryant, D.: Application of phylogenetic networks in evolutionary studies. Molecular Biology and Evolution 23(2), 254–267 (2006)
17. Lynce, I., Marques Silva, J.P.: Efficient haplotype inference with boolean satisfiability. In: Proceedings of National Conference on Artificial Intelligence (AAAI) (2006)
18. Moret, B., Nakhleh, L., Warnow, T., Linder, C.R., Tholse, A., Padolina, A., Sun, J., Timme, R.: Phylogenetic networks: Modeling, reconstructibility and accuracy. IEEE Transactions on Computational Biology and Bioinformatics 1(1), 13–23 (2004)

19. Munzner, T., Guimbrètiere, F., Tasiran, S., Zhang, L., Zhou, Y.: TreeJuxtaposer: Scalable tree comparison using Focus+Context with guaranteed visibility. In: SIGGRAPH 2003 Proceedings, published as special issue of Transactions on Graphics, pp. 453–462 (2003)
20. Nakhleh, L., Ruths, D., Wang, L.-S.: RIATA-HGT: A fast and accurate heuristic for reconstructing horizontal gene transfer. In: Wang, L. (ed.) COCOON 2005. LNCS, vol. 3595, pp. 84–93. Springer, Heidelberg (2005)
21. Olsen, G.J., Matsuda, H., Hagstrom, R., Overbeek, R.: Fastdnaml: A tool for construction of phylogenetic trees of dna sequences using maximum likelihood. Comput. Appl. Biosci. 10, 41–48 (1994)
22. Sanderson, M.J.: r8s; inferring absolute rates of evolution and divergence times in the absence of a molecular clock. Bioinformatics 19, 301–302 (2003)
23. Schmidt, H.A.: Phylogenetic trees from large datasets. PhD thesis, Heinrich-Heine-Universitat, Dusseldorf (2003)
24. Selman, B., Kautz, H.A., Cohen, B.: Software, http://www.cs.rochester.edu/u/kautz/walksat/
25. Selman, B., Kautz, H.A., Cohen, B.: Local search strategies for satisfiability testing. In: Trick, M., Johnson, D.S. (eds.) Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability, Providence RI (1993)
26. Semple, C.: Hybridization networks. New Mathematical Models for Evolution. Oxford University Press, Oxford (2007)
27. Tompkins, D.A.D., Hoos, H.H.: UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 306–320. Springer, Heidelberg (2005)
28. Wu, Y.: A practical method for exact computation of subtree prune and regraft distance. Bioinformatics 25(2), 190–196 (2009)
29. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla:portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research 32, 565–606 (2008)

Finding Lean Induced Cycles in Binary Hypercubes*

Yury Chebiryak[1], Thomas Wahl[1,2], Daniel Kroening[1,2], and Leopold Haller[2]

[1] Computer Systems Institute, ETH Zurich, Switzerland
[2] Computing Laboratory, Oxford University, United Kingdom

Abstract. Induced (chord-free) cycles in binary hypercubes have many applications in computer science. The state of the art for computing such cycles relies on genetic algorithms, which are, however, unable to perform a complete search. In this paper, we propose an approach to finding a special class of induced cycles we call *lean*, based on an efficient propositional SAT encoding. Lean induced cycles dominate a minimum number of hypercube nodes. Such cycles have been identified in Systems Biology as candidates for stable trajectories of gene regulatory networks. The encoding enabled us to compute lean induced cycles for hypercubes up to dimension 7. We also classify the induced cycles by the number of nodes they fail to dominate, using a custom-built All-SAT solver. We demonstrate how *clause filtering* can reduce the number of blocking clauses by two orders of magnitude.

1 Introduction

Cycles through binary hypercubes have applications in numerous fields in computing. The design of algorithms that reason about them is an active area of research. This paper is concerned with obtaining a subclass of these cycles with applications in Systems Biology.

Biochemical reactions in gene networks are frequently modeled using a system of piece-wise linear ordinary differential equations (PLDE), whose number corresponds to the number of genes in the network [4]. It is of critical importance to obtain *stable* solutions, because only stable orbits describe biologically relevant dynamics of the genes. We focus on Glass PLDE, a specific type of PLDE that simulates neural and gene regulatory networks [7].

The phase flow of Glass networks spans a sequence of coordinate orthants, which can be represented by the nodes of a binary hypercube. The orientation of the edges of the hypercube is determined by the choice of focal points of the PLDE. The orientation of the edge shows the direction of the phase flow

* A part of this work was presented at the 7th Australia – New Zealand Mathematics Convention, Christchurch, New Zealand, December 11, 2008. The work was supported by ETH Research Grant TH-19 06-3.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 18–31, 2009.

at the coordinate plane separating the orthants. Thus, the paths in oriented binary hypercubes serve as a discrete representation of the continuous dynamics of Glass gene regulatory networks. A special kind of such paths, *coil-in-the-box* codes, is used for the identification of stable periodic orbits in the Glass PLDE. Coil-in-the-box codes with maximum length represent the networks with longest sequence of gene states for a given number of genes [10].

If a cycle in the hypercube is defined by a coil-in-the-box code, the orientation of all edges adjacent to the cycle can be chosen to direct the flow towards it (the cycle is then called a *cyclic attractor*). Such orientation ensures the convergence of the flow to a periodic attractor that lies in the orthants included in the path. If a node of the hypercube is not adjacent to the cycle, the node does not have edges adjacent to the cycle, and the orientation of the edges at this node does not affect the stability of the flow along the orthants that are defined by the coil-in-the-box code. The choice of edge orientation in turn is linked to the specification of focal points of the PLDE. Therefore, the presence of nodes that are not dominated indicates that the phase flow along the attractor is robust to any variations of the coefficients that define the equations in the orthant corresponding to that node [20]. We say that a node that is not dominated by the cycle is *shunned* by the cycle.

The computation of (preferably long) induced (i.e., chord-free) cycles that dominate as few nodes as possible is therefore highly desirable in this context. We call such cycles *lean induced cycles*.

The state-of-the art in computing longest induced cycles and paths relies on genetic algorithms [5]. However, while this technique is able to identify individual cycles with desired properties, it cannot guarantee completeness, i.e., it may miss specific cycles. Many applications, including those in Systems Biology, rely on a classification of *all* solutions, which precludes the use of any incomplete random search technique.

Recent research suggests that SAT-based algorithms can solve many combinatorial problems efficiently: applications include oriented matroids [18], the coverability problem for unbounded Petri nets [1], bounds on van der Waerden numbers [12,6], and many more. Solving a propositional formula that encodes a desired combinatorial object with a state-of-the-art SAT solver can be more efficient than the alternatives.

Contribution. We encode the problem of identifying lean induced cycles in binary hypercubes as a propositional SAT formula and compute solutions using a state-of-the-art solver. As we aim at the complete set of cycles, we modify the solver to solve the All-SAT problem, and present three orthogonal optimizations that reduce the number of required blocking clauses by two orders of magnitude.

Our implementation enabled us to obtain a broad range of new results on cycles of this kind. L. Glass presented a coil-in-the-box code with one shunned node in the 4-cube [10]. We show that this is the maximum number of shunned nodes that any *lean* induced cycle may have for that dimension. Then, we show

that the longest induced cycles in the next two dimensions are *cube-dominating*: these cycles dominate every node of the cube. In dimension 7, where an induced cycle can be almost twice as long as the shortest cube-dominating cycles, there are lean induced cycles shunning at least three nodes.

2 Preliminaries

We define basic concepts used frequently throughout the paper. The *Hamming distance* between two bit-strings $u = u_1 \dots u_n$, $v = v_1 \dots v_n \in \{0,1\}^n$ of length n is the number of bit positions in which u and v differ:

$$d_H^n(u,v) = |\{\, i \in \{1,\dots,n\} \,:\, u_i \neq v_i \,\}|\,.$$

The *n-dimensional Hypercube*, or *n-cube* for short, is the graph (V,E) with $V = \{0,1\}^n$ and $(u,v) \in E$ exactly if $d_H^n(u,v) = 1$ (see also [14]). The n-cube has $n \cdot 2^{n-1}$ edges. We use the standard definitions of *path* and *cycle* through the hypercube graph. The length of a path is the number of its vertices. A Hamiltonian path (cycle) through the n-cube is called a *(cyclic) Gray code*. The *cyclic distance* of two nodes W_j and W_k along a cycle of length L in the n-cube is

$$d_C^n(W_j, W_k) = \min\{|k-j|,\ L - |k-j|\}\,.$$

In this paper, we are concerned with particular cycles through the n-cube.

Definition 1. *An **induced cycle** $I_0 \dots I_{L-1}$ in the n-cube is a cycle such that any two nodes on the cycle that are neighbors in the n-cube are also neighbors in the cycle:*

$$\forall j,k \in \{0,\dots,L-1\} \quad (d_H^n(I_j,I_k) = 1 \ \Rightarrow\ d_C^n(I_j,I_k) = 1)\,. \tag{1}$$

Fig. 1 shows an induced cycle (bold edges) in the 4-cube. In this paper, we are also interested in the immediate neighborhood of the cycle:

Definition 2. *The cycle $I_0 \dots I_{L-1}$ **dominates** node W of the n-cube if W is adjacent to some node of the cycle:*

$$\exists j \in \{0,\dots,L-1\} \quad d_H^n(I_j, W) = 1\,. \tag{2}$$

We say the cycle *shuns* the nodes it does not dominate. A cycle is called *cube-dominating* if it dominates every node of the n-cube; such cycles can be thought of as "fat". In contrast, in this paper we are interested in "lean" induced cycles, which dominate as few nodes as possible:

Definition 3. *A **lean** induced cycle is an induced cycle through the n-cube that dominates a minimum number of cube nodes, among all induced n-cube cycles **of the same length**.*

Especially significant are induced cycles of maximum length. The induced cycle in Fig. 1 is longest (length 8) in dimension 4. It is also lean, as it dominates 15 of the 16 cube nodes, and there is no induced cycle of length 8 dominating less than 15 nodes.

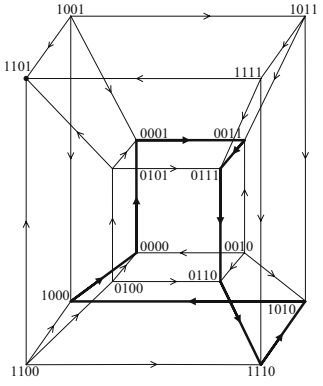


Fig. 1. A lean induced cycle in the 4-cube. The cycle shuns node 1101.

Lean induced cycles in cell biology. Hypercubes with lean induced cycles can aid the synthesis of Glass Boolean networks with stable periodic orbits and stable equilibrium states. For example, *C. elegans* vulval development is known to exhibit a series of cell divisions with 22 nuclei formed in the end of the development. The cell division represents a complex reactive system and includes at least four different molecular signaling pathways [15]. If the state of every signaling pathway is represented by a valuation of a Boolean variable, the 4-cube in Fig. 1 is useful for synthesizing a Glass Boolean network with a stable periodic orbit describing the cell division and an equilibrium depicting the finale state (at node 1101) of the gene regulatory system.

Co-existence of an induced cycle of maximum length and a shunned node in a hypercube indicates that during cell division, the gene network may traverse the maximum possible number of the different states before switching to the final equilibrium.

3 Computing Lean Induced Cycles

In this section, we describe an encoding of induced cycles of a given length into a propositional-logic formula. We then strengthen the encoding to assert the existence of a certain number of shunned nodes. We finally illustrate how we used the MiniSat solver to determine lean induced cycles where this number of shunned nodes is maximized.

3.1 A SAT-Encoding of Induced Cycles with Shunned Nodes

Our encoding relies heavily on comparing the Hamming distance between two hypercube nodes against some constant. We implement such comparisons efficiently using *once-twice* chains, as described in [3]. In brief, a once-twice chain

identifies differences between two strings up to some position j based on (i) comparing them at position j, and (ii) recursively comparing their prefixes up to position $j-1$.

Induced Cycles. We use $n \cdot L$ Boolean variables $I_j[k]$, where $0 \leq j < L$ and $0 \leq k < n$, to encode the coordinates of an induced cycle of length L in the n-cube. The variable $I_j[k]$ denotes the k-th coordinate of the j-th node. In order to form a cycle in an n-cube, consecutive nodes of the sequence must have Hamming distance 1, including the last and the first:

$$\varphi_{cycle} \quad := \quad \left(\bigwedge\nolimits_{i=0}^{L-2} d_H^n(I_i, I_{i+1}) = 1 \right) \quad \wedge \quad d_H^n(I_{L-1}, I_0) = 1 \,.$$

To make the cycle induced, we eliminate chords as follows:

$$\varphi_{chord\text{-}free} \quad := \quad \bigwedge_{\substack{0 \leq i < j < L, \\ d_C^n(I_i, I_j) \geq 2}} d_H^n(I_i, I_j) \geq 2 \,.$$

This also ensures that the nodes along the cycle are pairwise distinct. In practice, the formula $\varphi_{chord\text{-}free}$ can be optimized by eliminating half of its clauses, using an argument presented in [2].

The conjunction of these constraints is an encoding of induced cycles:

$$\varphi_{IC} \quad := \quad \varphi_{cycle} \ \wedge \ \varphi_{chord\text{-}free} \,.$$

Shunned Nodes. We encode the property that a cycle $I_0 \ldots I_{L-1}$ shuns nodes $u_0, \ldots, u_{S-1}$, by requiring the distance of the nodes to the cycle to be at least 2:

$$\varphi_{shunned} \quad := \quad \bigwedge\nolimits_{i=0}^{S-1} \bigwedge\nolimits_{j=0}^{L-1} d_H^n(u_i, I_j) \geq 2 \,.$$

We combine this with the condition that the nodes are distinct,

$$\varphi_{distinct} \quad := \quad \bigwedge\nolimits_{0 \leq i < j < S} d_H^n(u_i, u_j) \geq 1 \,,$$

to obtain an encoding of induced cycles with at least S shunned nodes:

$$\varphi_{ICS} \quad := \quad \varphi_{IC} \ \wedge \ \varphi_{shunned} \ \wedge \ \varphi_{distinct} \,. \tag{3}$$

We point out some basic monotonicity properties of formula φ_{ICS}. Let $IC(n, L, S^+)$ be the number of induced cycles of length L in the n-cube with at least S shunned hypercube nodes. It is easy to see that

$$\begin{array}{lcl} n_1 \leq n_2 & \Rightarrow & IC(n_1, L, S^+) \ \leq \ IC(n_2, L, S^+) \,, \quad \text{and} \\ S_1 \leq S_2 & \Rightarrow & IC(n, L, S_1^+) \ \geq \ IC(n, L, S_2^+) \,. \end{array}$$

There is no analogous monotonicity law for the length parameter L of an induced cycle. Intuitively, a medium value for L provides the greatest degree of freedom for a cycle.

Table 1. Length of longest induced cycles, and number of shunned nodes

dim. n	length L	max. # shunned nodes
3	6	0
4	8	1
5	14	0
6	26	0
7	48	≥ 3

3.2 Computing Lean Induced Cycles Using a SAT Solver

Every solution to equation (3) corresponds to an induced cycle of length L in the n-cube with at least S shunned nodes. In order to make the cycle *lean*, we need to maximize S. We achieve this by starting with *cube-dominating* induced cycles, i.e., with $S = 0$, and increasing S in equation (3) until the SAT solver reports unsatisfiability.[1]

Table 1 shows our findings for hypercubes up to dimension 7. For the classical cube of dimension 3, the longest induced cycles have length 6. All of those are cube-dominating. In dimension 4, the longest induced cycles have length 8; an example is shown in Fig. 1. Some of these cycles shun 1 of the 16 cube nodes; the others are cube-dominating. Interestingly, in dimensions 5 and 6, all longest induced cycles are again cube-dominating.

In dimension 7, we found longest (length 48) induced cycles shunning 3 nodes. For larger values of S, our search timed out after 24h. In our experiments, we used the MiniSat solver by Eén and Sörensson [9]. MiniSat provides interfaces for incremental solving and All-SAT; the current version uses preprocessing techniques [8] that simplify the original formula. All experiments were carried out on an Intel Xeon 3.0 GHz, 4-GB RAM PC running Linux.

4 Classification of Induced Cycles

The goal of this section is to determine how many distinct induced cycles of length L and with S shunned nodes exist in the n-cube, for a given triple (n, L, S). By *distinct*, we mean that the cycles cannot be transformed into each other by applying a symmetry permutation of the n-cube. That is, for each tuple (n, L, S), we *classify* the induced cycles into equivalence classes.

The classification of induced cycles with respect to symmetries of a hypercube is of interest in Glass models for neural and gene regulatory networks, because the number of the equivalence classes of the codes indicates how many different types of cells can be regulated by a set of genes [21,10].

[1] Since the range of values for S for which (3) is satisfiable is contiguous, a binary search strategy is also possible, using a heuristically determined initial value for S.

The enumeration of the equivalence classes is achieved using a custom-made All-SAT solver derived from MiniSat. We introduce blocking clauses that suppress solutions symmetric to one encountered before. We observe that cycles identical up to cube symmetries belong to the same class (n, L, S). This ensures that the symmetry breaking does not eliminate solutions with a different set of parameters. In the rest of this section, we describe the classification and the symmetry breaking in more detail.

4.1 Identifying Equivalence Classes Using Coordinate Sequences

In order to identify symmetry equivalence classes of cycles, it proved efficient to encode cycles in a slightly different way.

Definition 4 ([10]). *The* ***coordinate sequence*** *of a cycle $I_0 \dots I_{L-1}$ in the n-cube is the sequence $(c_0, \dots, c_{L-1}) \in \{0, \dots, n-1\}^L$ such that c_i is the unique coordinate that distinguishes I_i and $I_{i+1 \bmod L}$.*

For example, the coordinate sequence of the cycle in Fig. 1 is the sequence $cs := (0,\ 1,\ 2,\ 0,\ 3,\ 2,\ 1,\ 3)$, assuming $I_0 = 0000$ and $I_1 = 0001$. The dimensions are listed in the order 3210 in the figure.

Given coordinate sequences, we can define cube symmetries.

Definition 5. *Two cycles C_1 and C_2 in the n-cube are* ***equivalent****, $C_1 \sim C_2$, if their coordinate sequences are identical up to axis permutations, reflections about the center position, and rotations by an arbitrary number of coordinates.*

Given n and L, let CS denote the set of coordinate sequences of cycles of length L in the n-cube. A reflection or rotation on CS is a permutation π on the set $\{0, \dots, L-1\}$ that maps a coordinate sequence $(c_i)_{i=0}^{L-1}$ to the sequence $(c_{\pi(i)})_{i=0}^{L-1}$, that is, by acting on the position indices of the sequence. In contrast, an axis permutation on CS is a permutation π on the set $\{0, \dots, n-1\}$ that maps a coordinate sequence $(c_i)_{i=0}^{L-1}$ to the sequence $(\pi(c_i))_{i=0}^{L-1}$, that is, by acting on the coordinate values of the sequence. For example, the coordinate sequence $cs' := (1,\ 0,\ 2,\ 3,\ 0,\ 1,\ 3,\ 2)$ is equivalent to sequence cs above, since cs' can be obtained from cs by a left-rotation by one position, followed by a reflection and an axis permutation (1 2 3 0), mapping 1 to 2, 2 to 3, etc.

Our goal is to classify induced cycles based on cube symmetries, for a given parameter tuple (n, L, S). In order for this classification to be sound, the symmetry permutations must not alter the (n, L, S) parameters of a cycle.

Lemma 1. *Let C_1 and C_2 be two equivalent cycles. Then C_1 and C_2 have the same length and shun the same number of cube nodes.*

Proof (sketch). Since C_1 and C_2 are equivalent, there is a sequence Π of permutations, of the type mentioned in definition 5, such that $\Pi(C_1) = C_2$. Reflections and rotations of the coordinate sequence of C_1 translate to reversals of C_1's orientation, and to rotations of C_1, respectively. These operations change neither the length of the cycle, nor the distance of cube nodes to it.

For an axis permutation π, we have to show that definition 2, *dominates*, is invariant under π. We omit the technical derivation of this property. □

As an example, the unique cycle of the 4-cube corresponding to the above-mentioned coordinate sequence cs', after fixing $I_0 := 0000$ and $I_1 := 0010$, is lean and induced, as is the cycle in Fig. 1. Both cycles shun one cube node. Conversely, cycles with the same parameters (n, L, S) may not be equivalent: Table 2 (see Appendix) lists two distinct – in the above sense – cycles with $(n, L, S) = (4, 8, 0)$.

We determine the number $IC(n, L, S)$ of $\sim$ equivalence classes of induced cycles of length L with *exactly* S shunned nodes as the difference between the number of classes of cycles shunning at least S and $S + 1$ nodes, respectively:

$$IC(n, L, S) \quad = \quad IC(n, L, S^+) \; - \; IC(n, L, (S + 1)^+) \, . \tag{4}$$

The quantities on the right are computed, separately for S and $S + 1$, by enumerating satisfying assignments to Eq. (3), using an All-solutions SAT solver, implemented on top of MiniSat (see Algorithm 1 on the next page).

As proposed in [3], we encode coordinate sequences using XOR gates on Boolean variables denoting coordinates of a cycle. We write $xor^k[m]$ to refer to the m-th bit in bitwise xor-operation over coordinates of nodes I_k and $I_{k+1 \mod L}$. For example, if $xor^3[2]$ evaluates to true, dimension 2 is traversed while going from I_3 to I_4. We call the variables $xor^k[m]$ the "xor-variables".

To ensure a single representative for each $\sim$ equivalence class, we add blocking clauses for each solution found that prevent permutations of axes, rotations and reflections of the coordinate sequence of the solution. The number of blocking clauses to add per solution is $(2L \cdot n!)$. This is clearly a computational burden for the SAT solver, especially when the solution space is nearly exhausted, and the All-SAT procedure is about to find the formula to be unsatisfiable. In the rest of this section, we present techniques that reduce both the number and the length of the blocking clauses.

4.2 Optimizations

Compressing blocking clauses. A blocking clause for a given induced cycle, barring permutations of axes and rotations/reflections of a coordinate sequence, is expressed in terms of the variables encoding the sequence. For instance, to block permutations of the cycle in Fig. 1, we add the following clause:

$$\begin{aligned}
&(\neg xor^0[0] \vee xor^0[1] \vee xor^0[2] \vee xor^0[3] \\
&\vee xor^1[0] \vee \neg xor^1[1] \vee xor^1[2] \vee \; xor^1[3] \\
&\vdots \\
&\vee xor^7[0] \vee xor^7[1] \vee xor^7[2] \vee \neg xor^7[3] \;) \, .
\end{aligned}$$

Algorithm 1: COMPUTE-EQUIVALENCE-CLASSES

Input: the SAT instance I with fixed n, L, S;
the equivalence relation $\sim$
Output: The set of equivalence classes EC

```
1: EC := {}
2: SAT_solver.solve(I)
3: while SAT
4:   do IC = SAT_solver.decode()
5:      EC ← EC ∪ {IC}
6:      ∀IC_j ∼ IC.  I.add_blocking_clause(IC_j)
7:      SAT_solver.solve(I)
```

The length of this blocking clause is $(n \cdot L)$. Our first, and simplest, optimization is to omit literals that evaluate to *false*, since we know that these variables encode unit Hamming distance:

$$(\neg xor^0[0] \vee \neg xor^1[1] \vee \neg xor^2[2] \vee \neg xor^3[0] \vee \dots) .$$

This reduces the length of a clause to L.

Symmetric Cycles. The following optimization applies to specific cycles, called *symmetric induced cycles*. A Gray code is *symmetric*[2] if elements of its coordinate sequence that are $L/2$ apart are identical [19]. For a symmetric induced cycle, the number of blocking clauses to be added can be reduced by one-half: rotations by more than $L/2$ positions result in cycles that were already blocked.

Prefix Filtering. Without loss of generality, we fix the first two elements of the coordinate sequence to $(0, 1)$. For the next coordinate, dimension 0 cannot be traversed because this would form a chord. Neither can dimension 1, since the cycle must be simple. Out of higher dimensions, we can restrict the search to the *canonical class*[3] with prefix $(0, 1, 2)$. We enforce this prefix by fixing the values of the corresponding xor-variables using the following three clauses:

$$xor^0[0] \wedge xor^1[1] \wedge xor^2[2] . \tag{5}$$

This drastically reduces the number of solutions in each equivalence class, and eliminates a large number of blocking clauses. For example, it becomes unnecessary to add a blocking clause for the coordinate sequence cs' on page 24, as cs' is blocked by Eq. (5).

[2] This definition is not to be confused with the definition in [13], where this term refers to a code for which the number of bit changes is uniformly distributed among the bit positions, hence called a *balanced* Gray code in [17, p. 7].

[3] A canonical coordinate sequence is the one in which each coordinate k appears before the first appearance of $k + 1$ [11].

Phase Saving. In an attempt to speed up the enumeration of solutions, we added phase-saving [16] to MiniSat. By default, MiniSat assigns *false* to all decision variables. With phase saving, they are assigned their most recent values in the search. Phase saving combines well with aggressive restarting schemes, since it retains more information between restarts. Our intuition was that after finding a solution, the solver might be able to quickly identify neighboring solutions. Phase-saving alone, however, did not result in any speedups.

Ordering decision variables. Upon closer inspection of All-SAT runs, we found that the activity-based variable selection heuristic mainly chooses from a small set of branching variables. These variables correspond closely to the encoding of solutions in the input CNF. In order to make use of this insight, we extended the solver to allow for prioritization of important variables in the decision heuristic: In this modification, unprioritized variables are only considered for branching after all prioritized variables are assigned a value. We tested a number of possible restrictions, and found that prioritizing the variables that encode the induced-cycle nodes $I_0, \ldots, I_{L-1}$ works well for some instances, but yields bad results in general.

Combined Restart Policy. We found that the enumeration of solutions could be sped up by disabling the geometric restart scheme, but this led to bad performance on the final hard instances. By combining an initial high restart limit (100000 conflicts) with a subsequent switch to MiniSat's original geometric policy, starting again from a very low limit (100 conflicts), we were able to gain a 20% overall speed-up. Easier SAT instances can then be solved before the first restart, while hard instances still profit from aggressive restarts.

Further experiments with different combinations of the discussed strategies revealed that a combination of a high-restart limit, variable prioritization, and phase saving also led to a performance increase of about 20%.

4.3 Evaluation

Using prefix filtering and the optimizations for symmetric cycles, we are able to reduce the number of clauses drastically. As an example, consider an instance encoding induced cycles of length 26 in a 6-dimensional hypercube. In order to block a solution, we need to add only 312 blocking clauses in the non-symmetric case and 156 clauses for a symmetric cycle, instead of originally 37440. Our findings are presented in Fig. 2 and extend the classification presented in [22].

For some circuit length values L, the time required by the All-SAT solver increases with the number of shunned nodes. For such values of L, it is faster to perform the classification for a small value of S and then check how many nodes the cycles dominate.

In general, the time required to find the first induced cycle is a few orders of magnitude less than that to perform the classification, even in the case of one class only, as the run-time is dominated by the final unsatisfiable instance.

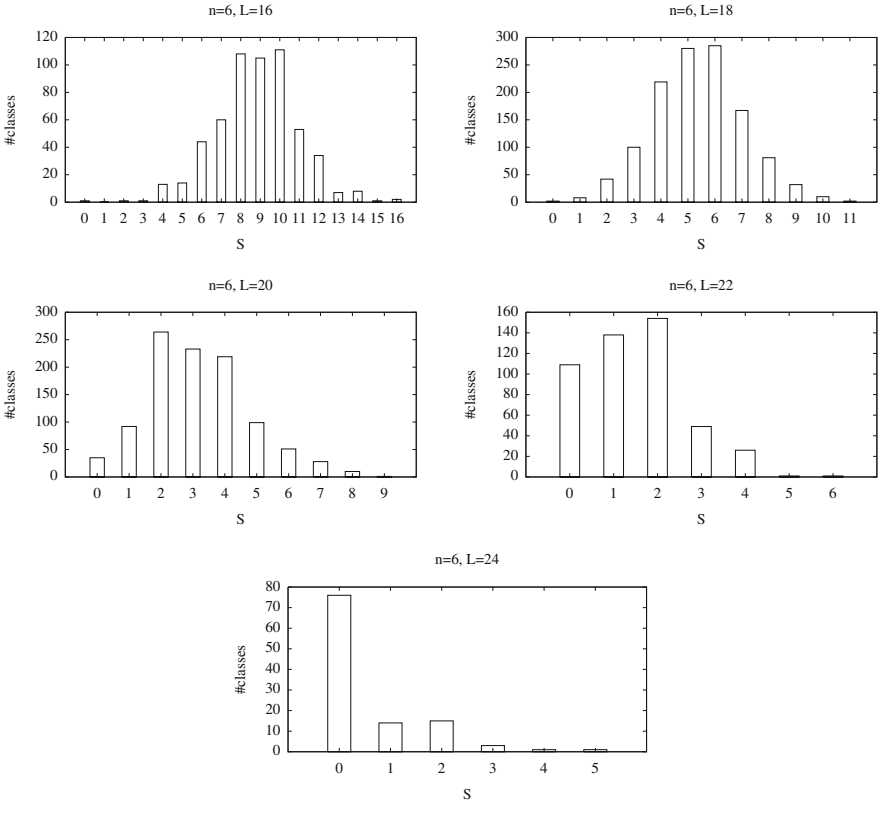


Fig. 2. Classification of induced cycles by cube symmetries, for select triples (n, L, S)

5 Conclusion

In this paper we have formalized a combinatorial problem relevant in Systems Biology: finding lean induced cycles in a hypercube, i.e., induced cycles that dominate a minimum number of hypercube nodes. We have presented a solution to this problem based on an efficient SAT encoding, and used this encoding to find lean induced cycles using a SAT solver. When compared to genetic algorithms, our method can provide guarantees for finding solutions, or prove the absence thereof.

Our method is suitable for classifying large sets of solutions into symmetry equivalence classes. As suggested by Fig. 2, this allows insights into the distribution of distinct solutions across the parameters n, L, and S. The SAT solver's performance is improved by filtering blocking clauses based on combinatorial properties of induced cycles, and by applying All-SAT specific internal tunings.

Acknowledgments

The authors would like to thank Dr. Igor Zinovik for bringing their attention to the problem of lean induced cycles and helping with preparing this script. They also thank the anonymous reviewers for suggestions on how to improve the draft.

References

1. Abdulla, P.A., Iyer, S.P., Nylén, A.: SAT-solving the coverability problem for Petri nets. Formal Methods in System Design 24(1), 25–43 (2004)
2. Chebiryak, Y., Kroening, D.: An efficient SAT encoding of circuit codes. In: Procs. IEEE International Symposium on Information Theory and its Applications, Auckland, New Zealand, December 2008, pp. 1235–1238 (2008)
3. Chebiryak, Y., Kroening, D.: Towards a classification of Hamiltonian cycles in the 6-cube. Journal on Satisfiability, Boolean Modeling and Computation (JSAT) 4, 57–74 (2008)
4. de Jong, H., Page, M.: Search for steady states of piecewise-linear differential equation models of genetic regulatory networks. IEEE/ACM Trans. Comput. Biology Bioinform. 5(2), 208–222 (2008)
5. Diaz-Gomez, P.A., Hougen, D.F.: Genetic algorithms for hunting snakes in hypercubes: Fitness function analysis and open questions. In: SNPD-SAWN 2006: Proceedings of the Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, Washington, DC, USA, pp. 389–394. IEEE Computer Society, Los Alamitos (2006)
6. Dransfield, M.R., Marek, V.W., Truszczynski, M.: Satisfiability and computing van der Waerden numbers. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 1–13. Springer, Heidelberg (2004)
7. Edwards, R.: Symbolic dynamics and computation in model gene networks. Chaos 11(1), 160–169 (2001)
8. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
9. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
10. Glass, L.: Combinatorial aspects of dynamics in biological systems. In: Landman, U. (ed.) Statistical mechanics and statistical methods in theory and applications, pp. 585–611. Plenum Press (1977)
11. Knuth, D.E.: The Art of Computer Programming. fascicle 2: Generating All Tuples and Permutations, vol. 4. Addison-Wesley Professional, Reading (2005)
12. Kouril, M., Franco, J.V.: Resolution tunnels for improved SAT solver performance. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 143–157. Springer, Heidelberg (2005)
13. Liu, X., Schrack, G.F.: A heuristic approach for constructing symmetric Gray codes. Appl. Math. Comput. 155(1), 55–63 (2004)
14. Livingston, M., Stout, Q.: Perfect dominating sets. Congressus Numerantium 79, 187–203 (1990)

15. Na'aman Kam, D., Kugler, H., Rami Marelly, A., Hubbard, J., Stern, M.: Formal modelling of C. elegans development. A scenario-based approach. Modelling in Molecular Biology, 151–174 (2004)
16. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
17. Savage, C.: A survey of combinatorial Gray codes. SIAM Review 39(4), 605–629 (1997)
18. Schewe, L.: Generation of oriented matroids using satisfiability solvers. In: Iglesias, A., Takayama, N. (eds.) ICMS 2006. LNCS, vol. 4151, pp. 216–218. Springer, Heidelberg (2006)
19. Singleton, R.C.: Generalized snake-in-the-box codes. IEEE Transactions on Electronic Computers EC-15(4), 596–602 (1966)
20. Zinovik, I., Chebiryak, Y., Kroening, D.: Cyclic attractors in Glass models for gene regulatory networks. IEEE Trans. Inf. Theory: Special Issue on Molecular Biology and Neuroscience (December 2009) (accepted)
21. Zinovik, I., Kroening, D., Chebiryak, Y.: An algebraic algorithm for the identification of Glass networks with periodic orbits along cyclic attractors. In: Anai, H., Horimoto, K., Kutsia, T. (eds.) Ab 2007. LNCS, vol. 4545, pp. 140–154. Springer, Heidelberg (2007)
22. Zinovik, I., Kroening, D., Chebiryak, Y.: Computing binary combinatorial Gray codes via exhaustive search with SAT-solvers. IEEE Transactions on Information Theory 54(4), 1819–1823 (2008)

6 Appendix

Table 2 shows runtimes, and number of equivalence classes of induced cycles found, for various values of (n, L, S).

Table 2. Classification of induced cycles, with runtimes

n	L	S	Time (sec)		#classes	
			first cycle	All-SAT	$IC(n, L, S^+)$	$IC(n, L, S)$
4	6	0	0.003	0.010	1	0
		1	0.006	0.017	1	0
		2	0.010	0.028	1	1
		3		0.031		0
	8	0	0.007	0.305	3	2
		1	0.009	0.048	1	1
		2		0.015		0
5	10	0	0.016	400.378	10	0
		1	0.017	419.881	10	0
		2	0.027	392.274	10	3
		3	0.031	370.277	7	3
		4	0.047	356.335	4	3
		5	0.043	210.137	1	0
		6	0.095	183.403	1	1
		7		167.397		0
	14	0	0.033	535.027	3	3
		1		3.012		0
6	16	0	0.02	486.37	563	1
		1	0.01	534.55	562	0
		2	0.03	481.12	562	1
		3	0.02	514.36	561	1
		4	0.04	501.77	560	13
		5	0.04	768.08	547	14
		6	0.04	3252.77	533	44
6	24	0	0.08	1183.50	110	76
		1	0.07	695.37	34	14
		2	0.30	689.22	20	15
		3	1.76	592.51	5	3
		4	5.65	1062.92	2	1
		5	26.56	1364.86	1	1
		6	-	1014.34		0
6	26	0	0.42	583.43	4	4
		1	-	750.39	0	0

Finding Efficient Circuits Using SAT-Solvers*

Arist Kojevnikov[1], Alexander S. Kulikov[2], and Grigory Yaroslavtsev[3]

[1] OneSpin Solutions GmbH
[2] St. Petersburg Department of Steklov Institute of Mathematics
[3] Academic Physics and Technology University of the RAS
{arist,kulikov,grigory}@logic.pdmi.ras.ru

Abstract. In this paper we report preliminary results of experiments with finding efficient circuits (over binary bases) using SAT-solvers. We present upper bounds for functions with constant number of inputs as well as general upper bounds that were found automatically. We focus mainly on MOD-functions. Besides theoretical interest, these functions are also interesting from a practical point of view as they are the core of the residue number system. In particular, we present a circuit of size $3n + c$ over the full binary basis computing MOD_3^n.

1 Introduction

In the recent years SAT solving became one of the main tools for formal verification [15]. In [6] it was proposed to use SAT in another very important area of the digital hardware production, namely in logical design synthesis. At the present time most electronic design automation tools (EDA) use algebraic manipulations [12] or binary decision diagrams (BDD) [10]. There are some successful experiments with genetic algorithms [5] and annealing optimizations [17]. See [7] for a survey.

Kamath at el. [6] propose a translation of the logical design synthesis problem to SAT. In [3] experiments with modern SAT solvers using this translation were reported. One of the advantages of this method is that it can also be used to prove lower bounds on Boolean functions, i.e., to prove that circuits of a given size do not exist. We use a similar reduction to CNF, however we are working with a more general computational model, namely with circuits over any binary basis.

It is known that finding efficient circuits is a difficult and important task. For many important functions there is a large gap between known lower and upper bounds. This shows that our current understanding of circuits is quite poor. As Williams notes [21] it might be helpful to know optimal circuits for such functions at least for small values of input size. Knowing this could help us to understand the structure of optimal circuits for general functions.

* The first two authors are supported in part by RFBR (grant 08-01-00640-a) and RAS Program for Fundamental Research ("Modern Problems of Theoretical Mathematics").

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 32–44, 2009.

In this paper we report results of experiments with finding efficient circuits using SAT-solvers. We focus mainly on circuit complexity of MOD-functions defined as follows:

$$\mathrm{MOD}^n_{K,k}(x_1,\ldots,x_n) = 1 \text{ iff } \sum_{i=1}^{n} x_i \equiv k \pmod{K}$$

(we omit k and/or n when they are not important). These are one of the simplest symmetric Boolean functions. Circuit complexity of these functions was studied by many researchers. Still, we know the exact circuit complexity for only a few values of K. Table 1 shows known lower and upper bounds for MOD^n_K in different computational models. There, by C and L we denote the circuit and formula complexity, respectively; B_2 is the full binary basis, $U_2 = B_2 \setminus \{\oplus, \equiv\}$. Interestingly, for formulas and circuits over the bases U_2 and B_2 it is known that the complexity of MOD^n_K, $K = 3$ or $K \geq 5$, is not less than the complexity of MOD^n_4. However, for none of these models it is known that MOD^n_3 or MOD^n_5 is strictly harder than MOD^n_4.

Table 1. Known lower and upper bounds on the complexity of MOD^n_K in different computational models

K		L_{U_2}	L_{B_2}	C_{U_2}	C_{B_2}
2	lower	$\Theta(n^2)$ [8]	n	$3n + c$ [18]	$n - 1$
	upper				
3	lower	$\Omega(n^2)$ [8]	$\Omega(n \log n)$ [4]	$4n + c$ [22]	$2.5n + c$ [19]
	upper	$O(n^{2.58})$ [1]	$O(n^2)$ [20]	$7n + o(n)$ [13]	$5n + o(n)$ [13]
4	lower	$\Theta(n^2)$ [8]	$\Omega(n \log n)$ [4]	$4n + c$ [22]	$2.5n + c$ [19]
	upper	$O(n^2 \log^2 n)$ [4]		$5n$ [22]	
≥ 5	lower	$\Omega(n^2)$ [8]	$\Omega(n \log n)$ [4]	$4n + c$ [22]	$2.5n + c$ [19]
	upper	$O(n^{4.57})$ [14]	$O(n^{3.13})$ [14]	$7n + o(n)$ [13]	$5n + o(n)$ [13]

Another motivation for studying MOD-functions is that a residue number system [9] is based on such functions. One of the main advantages of the residue number system is that additions, subtractions and multiplications are carry-free.

MOD-functions can be computed inductively. For example, the optimal circuit of size $2.5n + c$ for MOD^n_4 by Stockmeyer [19] is constructed from blocks consisting of 10 gates that sums 4 new variables with a residue number modulo 4, see Fig. 1. There, the bits z_0, z_1 encode the value of $\sum_{i=1}^{n} x_i \pmod 4$ as follows:

$$\sum_{i=1}^{n} x_i \pmod 4 = \begin{cases} 0, & \text{if } (z_0, z_1) = (0,0), \\ 1, & \text{if } (z_0, z_1) = (1,1), \\ 2, & \text{if } (z_0, z_1) = (1,0), \\ 3, & \text{if } (z_0, z_1) = (0,1). \end{cases}$$

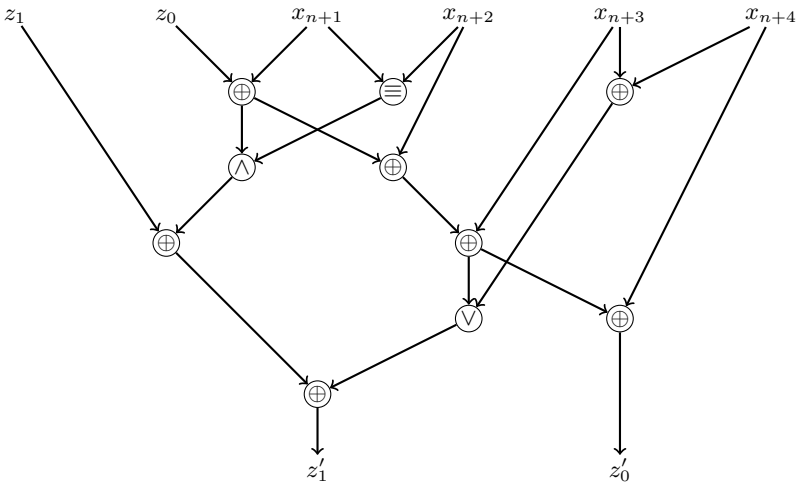


Fig. 1. Stockmeyer's block for MOD_4

The two output bits z'_0, z'_1 encode the value of $\sum_{i=1}^{n+4} x_i \pmod 4$ in the same way. Thus, one can prove general upper bounds on the circuit complexity of MOD-functions by finding efficient blocks of constant size. We report the results of experiments with finding such blocks by translating them to SAT.

The rest of the paper is organized as follows. In Sect. 2 we give all the necessary definitions. Section 3 describes the way we translate the fact of existence of a particular circuit into a CNF formula. In Sect. 4 we present some new circuit complexity upper bounds that were proved automatically. Sect. 5 presents results of experiments. Finally, in Sect. 6 we discuss some further directions.

2 General Setting

By B_n we denote the set of all Boolean functions $f\colon \{0,1\}^n \to \{0,1\}$. A function $f \in B_n$ is called symmetric if its value depends on the sum of the input bits only. That is, there must exist a vector $v \in \{0,1\}^{n+1}$ such that $f(x_1,\ldots,x_n) = v_s$ where $s = \sum_{i=1}^{n} x_i$. A typical symmetric function is a modular function $\mathrm{MOD}^n_{K,k}$ defined as follows:

$$\mathrm{MOD}^n_{K,k}(x_1,\ldots,x_n) = 1 \text{ iff } \sum_{i=1}^{n} x_i \equiv k \pmod K .$$

A circuit over the basis $A \subseteq B_2$ is a directed acyclic graph with nodes of in-degree 0 or 2. Nodes of in-degree 0 are marked by variables from $\{x_1,\ldots,x_n\}$ and are called inputs. Nodes of in-degree 2 are marked by functions from A and are called gates. There are also special output gates. The size of a circuit is its

number of gates. In this paper we mainly consider circuits over the full binary basis B_2.

We call a function $f \in B^n$ degenerated if it does not depend essentially on some of its variables, i.e., there is a variable x_i such that the subfunctions $f|_{x_i=0}$ and $f|_{x_i=1}$ are equal. It is easy to see that a gate computing a degenerated function from B_2 can be easily eliminated from a circuit without increasing its size (when eliminating this gate one may need to change the functions computed at its successors). For example, a gate computing NOT is degenerated. The set B_2 contains exactly ten non-degenerated functions $f(x, y)$:

- eight functions of the form $((x \oplus a) \wedge (y \oplus b)) \oplus c$, where $a, b, c \in \{0, 1\}$;
- two functions of the form $x \oplus y \oplus a$, where $a \in \{0, 1\}$;

3 Using SAT-Solvers for Finding Small Circuits

In this section we give the details of the reduction we use to encode the fact of existence of a particular circuit in CNF. We first describe the general reduction which is quite similar to the one described in [2] (where circuits over the basis U_2 are considered) and then discuss also some additional encoding which is specific to the considered functions.

3.1 Representing Circuits as CNFs

Given a truth table of a Boolean function $f: B^n \to B^m$ we would like to find a Boolean circuit computing f over the given basis $A \subseteq B_2$ with the smallest possible number of gates. We can encode the fact of existence of a circuit with N gates computing the function f in CNF using the following Boolean variables (input variables are numbered from 0 to $n-1$ and gates from n to $n+N-1$):

1. $t_{ib_1b_2}$ $(n \le i < n+N,\ 0 \le b_1 < 2,\ 0 \le b_2 < 2)$ is the output of the i-th gate if its first input is b_1 and the second is b_2. Four variables t_{i00}, t_{i01}, t_{i10}, t_{i11} thus completely define the binary Boolean function computed by the i-th gate. It gives $O(N)$ variables in total.
2. c_{ikj} $(n \le i < n+N-1,\ 0 \le k < 2,\ 0 \le j < n+N)$ is true if the k-th input of the i-th gate comes from the j-th gate and false otherwise. These variables completely define the underlying graph of a circuit. It gives $O(N^2)$ variables in total.
3. o_{ij} $(n \le i < n+N,\ 0 \le j < m)$ is true iff the j-th output of a circuit is computed by the i-th gate. These variables define which gates are used as outputs. It gives $O(Nm)$ variables in total.
4. v_{it} $(0 \le i \le n+N,\ 0 \le t < 2^n)$ is the output value of the i-th gate if the input variables have values represented by the bits of t. These variables are used to describe the fact that the values computed by a circuit are correct (according to the given truth table) on all 2^n assignments to input variables. It gives $O(2^n N)$ variables in total.

The following requirements about the circuit are written down as clauses.

1. Binary functions computed by gates belong to the basis A.
2. For all (i,k), exactly one variable c_{ikj} is true (the k-th input of the i-th gate is connected to only one gate). It gives $O(N^3)$ 2-clauses and $O(N)$ $O(N)$-clauses.
3. For all j, only one variable o_{ij} is true (the j-th output is computed by exactly one of the gates). It gives $O(N^2 m)$ 2-clauses and $O(m)$ $O(N)$-clauses.
4. For all $0 \le i < n$ and $0 \le t < 2^n$, v_{it} is equal to the corresponding bit in t. It gives $O(n \cdot 2^n)$ 1-clauses.
5. For all $n \le i < n+N$ and $0 \le t < 2^n$, v_{it} is equal to the value computed by the i-th gate in the part of a circuit described by other variables. It gives $O(N^3 \cdot 2^n)$ 6-clauses and this is where the most significant part of all clauses comes from. Clauses of this type are written for all $n \le i < n+N$, $n \le j_0 < i$, $j_0 < j_1 < i$, $0 \le i_0 < 2$, $0 \le i_1 < 2$, $0 \le r < 2^n$ and look as follows:
$$\neg c_{i0j_0} \vee \neg c_{i1j_1} \vee \neg(v_{j_0 r} = i_0) \vee \neg(v_{j_1 r} = i_1) \vee (v_{ir} = t_{ii_0 i_1}) .$$
Here first two literals let us find two gates, that are inputs of the i-th gate, the following two literals let us find the values of these gates on input assignment r and the last literal is for checking that the value in the i-th gate is correct.
6. The outputs of a circuit are computed correctly. It gives $O(N2^n m)$ 2-clauses. Clauses of this type are written for all $0 \le k < m$, $0 \le r < 2^n$, $n \le i < n+N$ and look as follows:
$$\neg o_{ik} \vee (v_{ir} = value_{kr}) ,$$
where $value_{kr}$ is the required value of the k-th output on input assignment r, according to the given truth table.

The clauses described above completely define all the requirements on a circuit. W.l.o.g. we can assume also that the following statements are true.

1. Both inputs of every gate are computed by gates with smaller numbers (i.e., the gates are sorted topologically w.r.t. the used numbering).
2. For every gate its first input gate has a smaller number than the second one.
3. The gates do not compute degenerated functions.
4. At least one of the outputs is computed by the last gate.

In most interesting cases the resulting formulas turn out to be quite difficult for modern SAT-solvers. E.g., a formula encoding the fact of existence of a circuit consisting of, say, 12 gates is already quite hard. This is because the number of different circuits as a function of the number of gates grows extremely fast. That is why in some cases we used also some additional restrictions that reduce the set of considered circuits. The main two of them are given below. Note however that unsatisfiability of a CNF formula with at least one of these restrictions does not imply that a circuit of a given size does not exist.

1. The out-degree of every gate is at most 2.
2. The i-th gate is fed by the $(i-1)$-th gate (i.e., there is a directed path through all the gates).

3.2 Residue Number Encodings

In the previous subsection we consider functions given by a truth table. Note however that one can work as well with partially defined functions and, more generally, with functions satisfying some particular properties. E.g., when searching for an inductive block for a MOD-function it is not clear what is the optimal encoding of a residue number. Thus, instead of providing a truth table of a block one can write down the fact that this block sums up new variables with a residue number which is encoded somehow.

Assume that we are looking for a block for MOD_K. Such a block sums up several variables with a residue number modulo K whose encoding is not known in advance. This residue number is encoded by $\lceil \log_2 K \rceil$ bits. Since the encoding is not known, we introduce new variables e_{ij}, where e_{ij} is true iff bit representation of $0 \le j < 2^{\lceil \log_2 K \rceil}$ encodes the residue number $0 \le i < K$. Thus, a particular residue number can be encoded by several j's. Except for some straightforward clauses stating that each i is encoded by some j and that each j is used for exactly one i we add also the following statement. For each possible assignment for inputs of a block, if the sum of input variables is s and the input residue number is i (i.e., the corresponding e_{ij} is true), then the output residue number cannot be j' for all j' such that $e_{i'j'}$ is true for some $i' \not\equiv i + s \pmod K$.

To give an example assume that we are searching for a block that takes as input a residue number t modulo 3 which is somehow encoded by two bits (z_0, z_1) and a new variable x_n and outputs two bits (z'_0, z'_1) that encode in the same way $t + x_n \pmod 3$. Then, e_{23} is true iff $(z_0, z_1) = (1, 1)$ implies $t = 2$ and e_{11} is true if $(z_0, z_1) = (0, 1)$ implies $t = 1$. Now we add the following constraint:

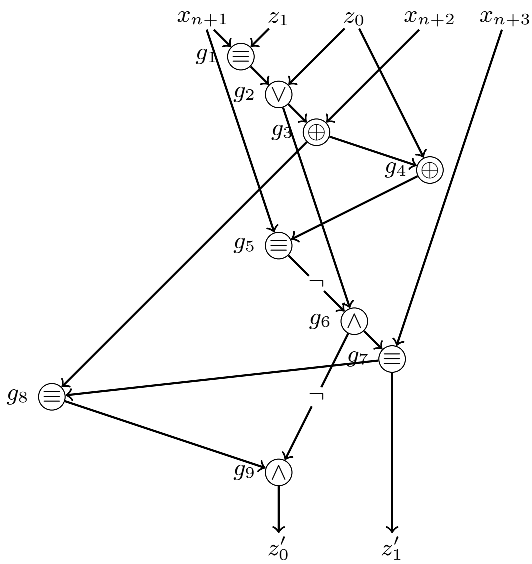
$$(e_{23} \wedge z_0 \wedge z_1 \wedge x_n \wedge e_{11}) \Rightarrow (z'_0 \vee \neg z'_1)\,.$$

These residue number encodings turned out to be quite helpful as only with them we were able to find an efficient block implying a $5.5n + c$ upper bound for $C_{U_2}(\mathrm{MOD}_3^n)$. However in most cases finding a block with an unknown encoding is a much more difficult task.

4 New Upper Bounds for MOD_3

Our main theoretical results are circuits of size $3n + O(1)$ and $5.5n + O(1)$ for MOD_3^n over the bases B_2 and U_2, respectively. The building blocks of the circuits (as well as their truth tables) are given in Fig. 2 and Fig. 3. The blocks take as input the value of $\sum_{i=1}^{n} x_i \pmod 3$ encoded by a pair of bits (z_1, z_2) and three respectively two new variables. The output is the pair of bits (z'_1, z'_2) encoding the value of $\sum_{i=1}^{n+3} x_i \pmod 3$ respectively $\sum_{i=1}^{n+2} x_i \pmod 3$. The residue number encodings for the blocks are the following:

$$\sum_{i=1}^{n} x_i \pmod 3 = \begin{cases} 0, & \text{if } (z_0, z_1) = (0, 0), \\ 1, & \text{if } (z_0, z_1) = (0, 1), \\ 2, & \text{if } z_0 = 1, \end{cases}$$

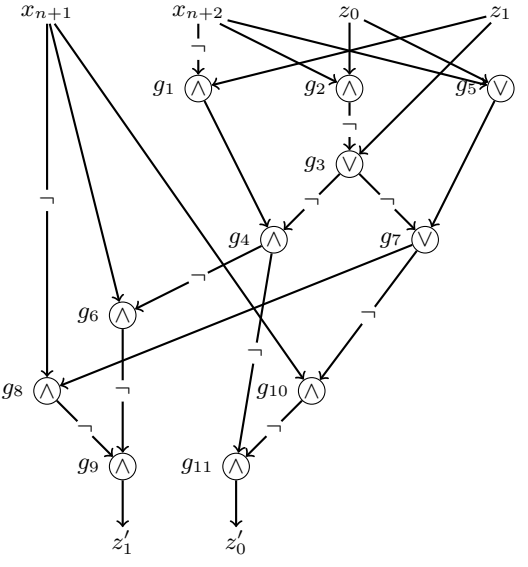


x_{n+1}	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
x_{n+2}	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x_{n+3}	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
z_0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
z_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
g_1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
g_2	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1
g_3	1	0	0	1	1	0	0	1	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0
g_4	1	0	0	1	1	0	0	1	0	0	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1
g_5	0	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	0	0	1	1	0	0	1
g_6	1	0	0	0	1	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	0	1	0	1	1	0	0	1	1	0
g_7	0	1	1	1	1	0	0	0	1	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	0	1	1	0
g_8	0	0	0	1	1	1	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	1	0	1
g_9	0	0	0	1	0	1	1	0	1	0	0	0	0	0	0	1	0	1	1	0	1	0	0	0	1	0	0	0	0	0	0	1
z'_0	0	0	0	1	0	1	1	0	1	0	0	0	0	0	0	1	0	1	1	0	1	0	0	0	1	0	0	0	0	0	0	1
z'_1	0	1	1	1	1	0	0	0	1	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	0	1	1	0

Fig. 2. An inductive block for MOD_3 over the basis B_2 and its truth table

and

$$\sum_{i=1}^{k} x_i \pmod 3 = \begin{cases} 0, & \text{if } z_0 = 0, \\ 1, & \text{if } (z_0, z_1) = (1, 0), \\ 2, & \text{if } (z_0, z_1) = (1, 1). \end{cases}$$



x_{n+1}	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
x_{n+2}	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
z_0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
z_1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
g_1	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0
g_2	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1
g_3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	1
g_4	1	1	0	0	1	1	0	0	0	0	0	0	0	0	1	1
g_5	0	0	1	1	1	1	1	1	0	0	1	1	1	1	1	1
g_6	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
g_7	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1
g_8	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
g_9	0	1	1	1	0	1	1	1	1	1	1	0	1	0	0	1
g_{10}	1	0	1	0	1	0	1	0	1	0	0	0	0	0	1	0
g_{11}	0	0	0	1	0	0	0	1	0	1	1	1	1	1	0	0
z'_0	0	1	1	1	0	1	1	1	1	1	1	0	1	0	0	1
z'_1	0	0	0	1	0	0	0	1	0	1	1	1	1	1	0	0

Fig. 3. An inductive block for MOD_3 over the basis U_2 and its truth table

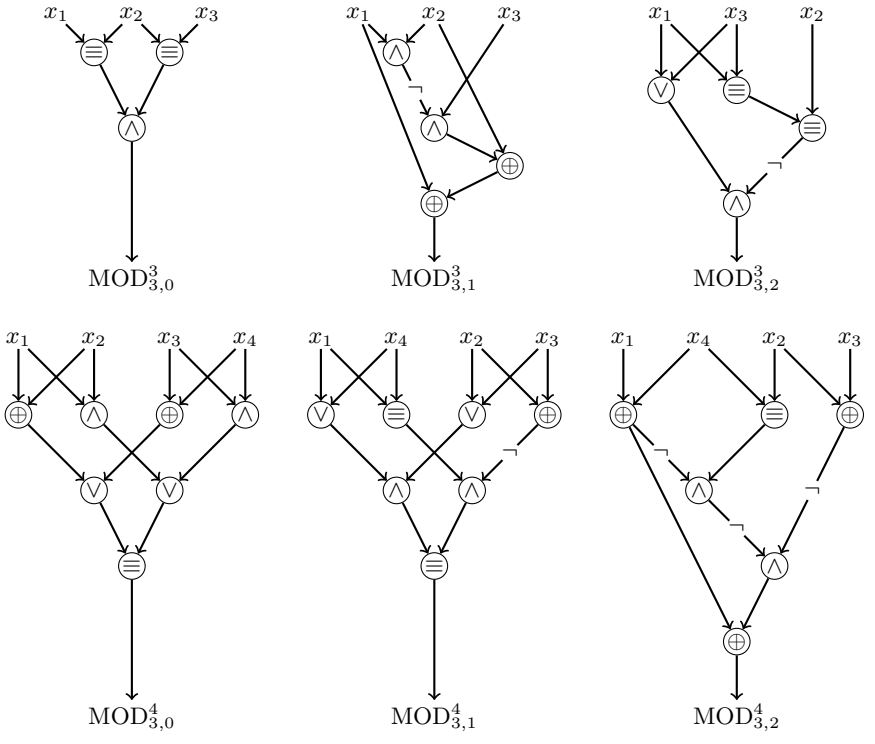


Fig. 4. Optimal circuits for $\mathrm{MOD}^n_{3,k}$ for $n = 3, 4$

The upper bounds $C_{B_2}(\mathrm{MOD}^n_3) \leq 3n + O(1)$ and $C_{U_2}(\mathrm{MOD}^n_3) \leq 5.5n + O(1)$ follow immediately from the existence of such blocks.

Table 2. Sizes of optimal circuits over B_2 for $\mathrm{MOD}^n_{3,k}$

	$n = 3$	$n = 4$	$n = 5$
$k = 0$	3	7	≤ 10
$k = 1$	4	7	≤ 9
$k = 2$	4	6	≤ 10

The blocks were found after a long sequence of experiments with different restrictions on considered circuits as without restrictions the resulting formulas cannot be handled by solvers. We still do not know whether the first block for adding three variables is optimal and thus a $8n/3$ upper bound for $C_{B_2}(\mathrm{MOD}^n_3)$ is not excluded.

Table 3. Statistics

	picosat	minisat207	oksolver	rsat	onespinsat	manysat
mod3_4vars_6gates.cnf	*	11m32s	*	*	3m16s	35m23s
mod4block_3vars_7gates.cnf	*	*	*	*	*	*
mod3block_3vars_9gates_restr.cnf	1m27s	0m5s	0m59s	0m2s	0m18s	0m29s
mod5block_1var_7gates.cnf	0m1s	0m2s	*	0m1s	0m1s	0m3s
mod4block_2vars_8gates_u2.cnf	*	*	*	*	22m43s	*
mod5block_2vars_12gates.cnf	*	*	*	*	*	*
mod4block_2vars_9gates_u2.cnf	*	*	*	*	*	*
owp_4vars_5gates.cnf	0m0.2s	0m0.3s	2m14s	0m0.3s	0m0.1s	0m0.7s

In Table 2 we also present the sizes of optimal circuits over B_2 for $\mathrm{MOD}^n_{3,k}$ for different values of n and k. $C_{B_2}(\mathrm{MOD}^5_{3,2}) \leq 10$ means that we found a circuit of size 10, but were unable to prove unsatisfiability of the fact that there exists a circuit (without any restrictions on its structure) of size 9. Optimal circuits are given in Fig. 4.

5 Empirical Studies

Table 3 provides the results of experiments with several formulas. All our experiments were made on a 2.40GHz AMD Opteron Processor 250 running under Linux. The DIMACS hardware benchmark program *dfmax r500.5* takes 7.11 seconds on the machine.

A two-hour limit applied to all runs of all solvers. Below we describe the benchmarks used in the table as well as benchmarks for which we still do not know the answer. By N, K and L we denote the number of variables, the number of clauses and the length (i.e., the total number of literals), respectively (however our formulas contain some unit clauses and can be simplified).

– SAT
  `mod3block_3vars_9gates_restr.cnf` ($N = 784$, $K = 219760$, $L = 1266513$) expresses the fact that there exists a circuit that sums up a residue number modulo 3 in a given encoding (0 — $(0,0)$, 1 — $(1,0)$, 2 — $(1,0),(1,1)$) with three new variables with two additional restrictions: there is a directed path through all the gates and the out-degree of any gate is at most 2. The corresponding circuit is given in Fig. 2.
  `mod5block_1vars_7gates.cnf` ($N = 353$, $K = 50103$, $L = 292994$) encodes the fact that there exists an inductive block for MOD^n_5 adding two new variables by 7 gates (residue number encoding is fixed).

– UNSAT
  `mod3_4vars_6gates.cnf` ($N = 289$, $K = 36396$, $L = 213400$) expresses the fact that $C_{B_2}(\mathrm{MOD}^4_{3,0}) \leq 6$.
  `owp_vars4_gates5.cnf` ($N = 267$, $K = 25667$, $L = 149802$) encodes the fact that $C_{B_2}(f) \leq 5$ for a permutation $f: B_4 \to B_4$ given in [11]. Unsatisfiability of this benchmark justifies the fact that f has asymmetric circuit complexity ($C_{B_2}(f) = 6$, $C_{B_2}(f^{-1}) = 5$).
– UNKNOWN
  `mod4block_3vars_7gates.cnf` ($N = 574$, $K = 125562$, $L = 742458$) encodes the fact that there exists an inductive block for MOD^n_4 adding three new variables by 7 gates (residue number encoding is fixed). Must be unsatisfiable, as otherwise MOD^n_4 could be computed by circuits of size about $7n/3$.
  `mod4block_2vars_{8,9}gates_u2.cnf` ($N = 387/426$, $K = 66496/86121$, $L = 389012/503636$) encodes the fact that there exists an inductive block for MOD^n_4 adding two new variables by $\{8, 9\}$ gates in the basis U_2 (residue number encoding is fixed).
  `mod3block_2vars_{9,10,11}gates_u2_autoenc.cnf` ($N = 426/475/526$, $K = 99345/125336/155313$, $L = 588054/741756/918948$) encodes the fact that there exists an inductive block for MOD^n_3 adding two new variables by 9 gates in the basis U_2. Here the residue number encoding is not fixed. Instead of this, benchmarks encode the fact that the required circuit sums up several bits with a residue number modulo 3 whose encoding is not known in advance (details are given in Sect. 3).
  `mod4block_2vars_{8,9,10,11}gates_u2_autoenc.cnf` ($383 \leq N \leq 530$, $81176 \leq K \leq 164070$, $480856 \leq L \leq 965444$) encodes the fact that there exists an inductive block for MOD^n_4 adding two new variables by $g = 8, 9, 10, 11$ gates in the basis U_2 (encoding is not fixed). Satisfiability of a benchmark from this family would imply that $C_{U_2}(\mathrm{MOD}^n_4) \leq gn/2 + c$ (note that at the moment it is only known that $4n - c \leq C_{U_2}(\mathrm{MOD}^n_4) \leq 5n + c$).

6 Further Directions

A natural further direction is to obtain more upper bounds for MOD_K-functions for different values of K. Note however that even if an optimal circuit for a MOD^n_K function can be constructed from inductive blocks, then these blocks must be large for large values of K, just because one needs many bits in order to encode a residue number modulo K. E.g., a block for summing up several new variables with a residue number modulo $K > 2^t$ must have at least t inputs and hence at least t gates. For $t \geq 15$, even finding such circuits is a really difficult task for modern SAT-solvers and proving that such a circuit does not exist is much more difficult.

It would be interesting also to find efficient circuits for other important functions. E.g., in [21] is is noted that it is easy to construct optimal circuits for 2×2-matrix multiplication, while already for 3×3 this is a difficult task. It would also

be interesting to know optimal circuits for small input sizes for the well-known CLIQUE-function which has super-polynomial complexity in the model of monotone circuits [16]. Knowing optimal circuits for a function on small input sizes could help to construct efficient circuits for all input sizes. SAT-solvers could also apparently help to improve current upper bounds for addition and multiplication. Note that the smallest known circuits and formulas for these functions are also built from blocks [14].

Using the described reduction one can produce different unsatisfiable formulas (e.g., encoding the fact that there exists a circuit of size smaller than the corresponding known lower bound). Such benchmarks turned out to be difficult for modern solvers. One could think about complexity of such benchmarks in different proof systems.

Another direction of further research is automatizing lower bounds proofs. Essentially the only known method for proving lower bounds for unrestricted circuits is gate elimination. For example, in order to prove a $2.5n-c$ lower bound for MOD_4^n Stockmeyer [19] proved that for any circuit computing $\mathrm{MOD}_{4,k}^n$ it is possible to eliminate five gates by assigning values to two input variables. The lower bound then follows by induction. However to prove that it is possible to eliminate five gates one needs to consider many different cases. It would be interesting to automate this case analysis.

Acknowledgments

The authors are very grateful to Anton Belov, Edward A. Hirsch, Evgeny Karibaev, Yakov Novikov and anonymous reviewers for helpful comments.

References

1. Chin, A.: On the depth complexity of the counting functions. Information Processing Letters 35, 325–328 (1990)
2. Eén, N.: Practical SAT — a tutorial on applied satisfiability solving. Slides of invited talk at FMCAD (2007)
3. Estrada, G.G.: A note on designing logical circuits using SAT. In: Tyrrell, A.M., Haddow, P.C., Torresen, J. (eds.) ICES 2003. LNCS, vol. 2606, pp. 410–421. Springer, Heidelberg (2003)
4. Fischer, M.J., Meyer, A.R., Paterson, M.S.: $\Omega(n \log n)$ lower bounds on length of Boolean formulas. SIAM Journal on Computing 11, 416–427 (1982)
5. Fogel, D.B.: Evolutionary computation: The fossil record. IEEE Press, New York (1998)
6. Kamath, A.P., Karmarkar, N.K., Ramakrishnan, K.G., Resende, M.G.C.: An interior point approach to boolean vector function synthesis. In: Proceedings of the 36th International Midwest Symposium on Circuits and Systems (MSCAS 1993), pp. 185–189 (1993)
7. Khatri, S., Shenoy, N.: Logic synthesis. In: Scheffer, L., Lavagno, L., Martin, G. (eds.) Electronic Design Automation For Integrated Circuits Handbook. CRC Press, Taylor & Francis Group (2006)

8. Khrapchenko, V.M.: Complexity of the realization of a linear function in the case of Π-circuits. Math. Notes Acad. Sciences 9, 21–23 (1971)
9. Koren, I.: Computer Arithmetic Algorithms. Prentice Hall, Englewood Cliffs (1993)
10. Lee, C.Y.: Representation of switching circuits by binary-decision programs. Bell Systems Technical Journal 38, 985–999 (1959)
11. Massey, J.L.: The difficulty with difficulty. In: A Guide to the Transparencies from the EUROCRYPT 1996 IACR Distinguished Lecture (1996)
12. McCluskey, E.J.: Logic Design Principles: with emphasis on testable semicustom circuits. Prentice-Hall, Englewood Cliffs (1986)
13. Nigmatullin, R.G.: Slognost' bulevikh funktsii. Moskva, Nauka (1991) (in Russian)
14. Paterson, M.S., Zwick, U.: Shallow circuits and concise formulae for multiple addition and multiplication. Computational Complexity 3, 262–291 (1993)
15. Prasad, M.R., Biere, A., Aarti, G.: A survey of recent advances in SAT-based formal verification. International Journal on Software Tools for Technology Transfer 7(2), 156–173 (2005)
16. Razborov, A.A.: Lower bounds for the monotone complexity of some Boolean functions. Soviet Math. Doklady 31, 354–357 (1985)
17. Gelatt, C.D., Kirkpatrick, S., Vecchi, M.P.: Optimization by simulated annealing. Science, New Series 220(4598), 671–680 (1983)
18. Schnorr, C.: Zwei lineare untere Schranken für die Komplexität Boolescher Funktionen. Computing 13, 155–171 (1974)
19. Stockmeyer, L.J.: On the combinational complexity of certain symmetric Boolean functions. Mathematical Systems Theory 10, 323–336 (1977)
20. van Leijenhorst, D.C.: A note on the formula size of the "mod k" functions. Information Processing Letters 24, 223–224 (1987)
21. Williams, R.: Applying practice to theory. ACM SIGACT News 39(4), 37–52 (2008)
22. Zwick, U.: A $4n$ lower bound on the combinational complexity of certain symmetric boolean functions over the basis of unate dyadic Boolean functions. SIAM Journal on Computing 20, 499–505 (1991)

Encoding Treewidth into SAT

Marko Samer and Helmut Veith

Department of Computer Science
TU Darmstadt, Germany
{samer,veith}@cs.tu-darmstadt.de

Abstract. One of the most important structural parameters of graphs is *treewidth*, a measure for the "tree-likeness" and thus in many cases an indicator for the hardness of problem instances. The smaller the treewidth, the closer the graph is to a tree and the more efficiently the underlying instance often can be solved. However, computing the treewidth of a graph is NP-hard in general. In this paper we propose an encoding of the decision problem whether the treewidth of a given graph is at most k into the propositional satisfiability problem. The resulting SAT instance can then be fed to a SAT solver. In this way we are able to improve the known bounds on the treewidth of several benchmark graphs from the literature.

1 Introduction

Many important combinatorial problems that are NP-hard in general are easy to solve if the instance is structured as a tree. *Tree decompositions* and the corresponding measure *treewidth* [13] gradually generalize the tree structure of instances to "tree-likeness". Roughly speaking, the smaller the treewidth, the less cyclic the graph and the closer the graph is to a tree. Instances with small treewidth can often be solved efficiently by dynamic programming on the tree decomposition. For example, Courcelle's Theorem [9] states that fixed graph properties expressible in monadic second-order logic (MSO) can be decided in linear time on graphs with bounded treewidth. However, for unbounded k, deciding whether a graph has treewidth at most k is NP-complete [1]. For this reason, the common way in practice for computing tree decompositions is the use of heuristics and approximation algorithms [2,4,8,12]. Although these methods often lead to tree decompositions of small width, the so obtained results are in general only upper bounds on the treewidth: the actual treewidth is the minimum width over all possible tree decompositions. Complementary to these upper bound methods, several approaches for computing lower bounds have been proposed as well [4,5,6,8,12]. Clearly, the treewidth of a graph has been found if the lower and upper bounds coincide. For many graphs, however, the lower and upper bound algorithms yield only an interval in which the treewidth is contained. To make these intervals smaller or to determine the treewidth, also exact algorithms based on branch-and-bound have been developed [3,11].

In this paper we propose one more algorithmic tool for determining the treewidth of graphs: We present an encoding of the decision problem whether a given graph $G = (V, E)$ has treewidth at most k to an instance F of the propositional satisfiability problem (SAT) such that G has treewidth less than or equal to k if and only if F is satisfiable. In this way we aim at exploiting the increasing power of modern SAT solvers to find

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 45–50, 2009.

tree decompositions of minimal width and thus to determine the treewidth of graphs. Our encoding results in SAT instances consisting of $\mathcal{O}(k\,|V|^2)$ variables and $\mathcal{O}(|V|^3)$ clauses; the length of each clause is bounded by 4. Using this approach, we have been able to determine the treewidth (or at least to shorten its interval) of several moderately sized benchmark graphs from the literature.

This paper is organized as follows: In Section 2, we formally introduce the notion of tree decompositions and treewidth. Moreover, we explain an alternative characterization of treewidth on which our encoding is based. Then, in Section 3, we present our encoding into propositional satisfiability and our experimental results. Finally, we conclude in Section 4.

2 Treewidth

A *graph* is a tuple (V, E) of a non-empty set V of vertices and a set $E \subseteq \{\{x, y\}\, x, y \in V\}$ of edges. A *tree decomposition* of a graph $G = (V, E)$ is a tree $T = (V', E')$ where each node $t \in V'$ is associated with a "bag" $\chi(t) \subseteq V$ containing vertices of G such that the following conditions are satisfied [13]:

1. $\bigcup_{t \in V'} \chi(t) = V$.
2. For every edge $e \in E$, there is some node $t \in V'$ such that $e \subseteq \chi(t)$.
3. For all nodes $t_1, t_2, t_3 \in V'$ such that t_2 lies on the (unique) path from t_1 to t_3 in T, it holds that $\chi(t_1) \cap \chi(t_3) \subseteq \chi(t_2)$.

The first condition requires that there are only vertices in the bags and every vertex occurs in some bag, the second condition requires that all pairs of vertices that are connected by an edge occur together in some bag, and the third condition ("connectedness condition") requires that if a vertex occurs in the bags of two different nodes, then it must occur in each bag on the unique path between them. The *width* of a tree decomposition $T = (V', E')$ is given by $\max_{t \in V'} |\chi(t)| - 1$. The *treewidth* of a graph is the minimum width over all its tree decompositions.

It is well known that from every linear ordering of the vertices of a graph $G = (V, E)$ one can easily construct a tree decomposition of G and that there is always a linear ordering such that the corresponding tree decomposition is optimal, i.e., its width equals the treewidth of G (see, e.g., Bodlaender [4] or Dechter [10]). Our encoding will be based on this alternative characterization. In fact, the linear ordering gives rise to a triangulation of the graph from which the corresponding width can be directly read off, i.e., we do not need to construct the tree decomposition explicitly. In this context, the treewidth is also called *induced width* [10]. Given a linear ordering $v_1, v_2, \ldots, v_n$ of the vertices in V, we call v_j a *predecessor* of v_i if $\{v_i, v_j\} \in E$ and $j < i$, and we call v_j a *successor* of v_i if $\{v_i, v_j\} \in E$ and $j > i$. To determine the width of the tree decomposition obtained from this linear ordering, we proceed as follows: For each pair of non-adjacent vertices v_i and v_j, we successively add an edge $\{v_i, v_j\}$ to E if and only if v_i and v_j have a common predecessor. This is repeated as long as new edges can be added. The width of the tree decomposition is then the maximum number of successors over all vertices, i.e., $\max_{v_i \in V} |\{\{v_i, v_j\} \in E : j > i\}|$. Figure 1 shows a

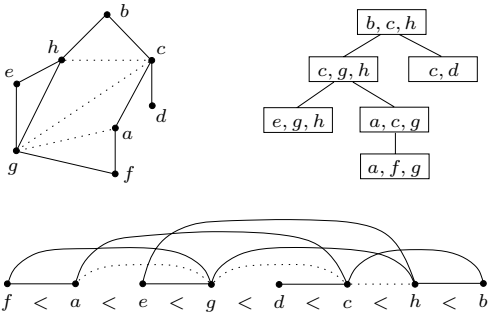


Fig. 1. A graph of treewidth 2, one of its tree decompositions of width 2, and a corresponding linear ordering (the dotted edges are added depending on the ordering)

graph, one of its tree decompositions, and the corresponding linear ordering. The solid edges belong to the input graph and the dotted edges are added according to the above rule: The edge $\{a, g\}$ is added because f is a predecessor of both a and g. This newly added edge now causes the edge $\{c, g\}$ to be added, which in turn causes the edge $\{c, h\}$ to be added. Since the maximum number of successors after this process is 2, we know that the corresponding tree decomposition has width 2.

3 The Encoding

Our encoding into propositional satisfiability is now based on the linear ordering as described in Section 2. Let $G = (V, E)$ be the input graph with $n = |V|$ and $m = |E|$ and assume that all vertices are enumerated from 1 to n. To encode the linear ordering itself, we introduce $n(n-1)/2$ Boolean variables $ord_{i,j}$ with $1 \leq i < j \leq n$: Variable $ord_{i,j}$ is true if and only if vertex v_i precedes vertex v_j in the linear ordering. In addition, we need $n(n-1)(n-2)$ clauses of the form

$$(ord^*_{i,j} \wedge ord^*_{j,l}) \rightarrow ord^*_{i,l}, \text{ where } ord^*_{p,q} = \begin{cases} ord_{p,q} & \text{if } p < q \\ \neg ord_{q,p} & \text{otherwise} \end{cases}$$

to enforce transitivity. Note that $ord^*_{p,q}$ is used here for presentation purposes only and is replaced in our encoding by $ord_{p,q}$ and $\neg ord_{q,p}$, respectively.

Having now encoded the linear ordering, we are going to encode the successor relation induced by this ordering. To this aim, note that the linear ordering gives a direction to all edges $\{v_i, v_j\} \in E$, namely from v_i to v_j if and only if v_j is a successor of v_i. Thus, we introduce n^2 Boolean variables $arc_{i,j}$ with $1 \leq i, j \leq n$: Variable $arc_{i,j}$ is true if vertex v_j is a successor of vertex v_i. We call such an arc variable *active* if it is assigned true and *inactive* otherwise. As mentioned in Section 2, the maximum number of successors, i.e., the maximum number of active outgoing arcs, is then the width of the corresponding tree decomposition. That means for each vertex we have to ensure by a cardinality constraint that the number of its active outgoing arcs does not exceed

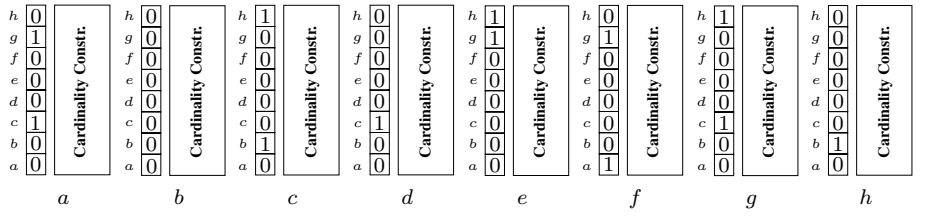


Fig. 2. Encoding of the successor relation induced by the linear ordering in Figure 1

our upper bound k on the treewidth. Figure 2 illustrates this encoding of the successor relation induced by the linear ordering in Figure 1.

Let us consider these steps in more detail. To encode the edges of the input graph, we add $2m$ clauses of the form $ord_{i,j} \to arc_{i,j}$ and $\neg ord_{i,j} \to arc_{j,i}$ for $\{v_i, v_j\} \in E$ and $i < j$. Moreover, in order to enforce the additional edges caused by the linear ordering (i.e., the dotted edges in Figure 1), we add $n(n-1)(n-2)$ clauses of the form

$$(arc_{i,j} \wedge arc_{i,l} \wedge ord_{j,l}) \to arc_{j,l} \quad \text{and} \quad (arc_{i,j} \wedge arc_{i,l} \wedge \neg ord_{j,l}) \to arc_{l,j}$$

for $1 \le i, j, l \le n$, $i \ne j$, $i \ne l$, and $j < l$. We also add $n(n-1)(n-2)/2$ clauses of the form $\neg arc_{i,j} \vee \neg arc_{i,l} \vee arc_{j,l} \vee arc_{l,j}$, which are actually redundant but accelerate the SAT solving process. In addition, since self loops can be neglected, we add n unit clauses of the form $\neg arc_{i,i}$ for $1 \le i \le n$.

We have now forced all outgoing arcs to be active if the corresponding vertex is a successor. To guarantee that the instance is satisfiable if and only if there exists a tree decomposition of width at most k, we have to make sure by a cardinality constraint that at most k outgoing arcs of each vertex are active. A naive way of doing this is to add $\binom{n}{k+1}$ clauses of length $k+1$ for each vertex to ensure that there is no subset of $k+1$ active outgoing arcs. This, however, would result in $\mathcal{O}(n^{k+1})$ additional clauses. For this reason, we implement the cardinality constraint by a counter that is able to count the number of active outgoing arcs up to k.[1] Our counter is a slightly improved variant of the sequential unary counter proposed by Sinz [14]. Figure 3 illustrates two different counting scenarios with our encoding. In both cases the column on the left represents the outgoing arc variables associated to a vertex. The boxes on the right represent auxiliary variables of the counter, where each row represents a number up to k in unary encoding. In particular, the counter works in a bottom-up manner such that the number encoded in the i-th row is (at least) the number of active arcs up to row i. Every time we find an active arc, the counter is incremented as indicted by the arrows. If a variable in the last column is assigned true, the counter has reached its upper bound and we add clauses that forbid further active arcs as indicated by the shaded boxes in the scenario on the right. Note that we neglect the topmost row of counter variables as $n-1$ rows are sufficient. In summary, each counter requires $k(n-1) - k(k-1)/2$ additional variables and $(2k+1)n - k(k+3)$ clauses, which can both be estimated by $\mathcal{O}(kn)$. We denote the counter variables by $ctr_{i,j,l}$ with $1 \le i \le n$, $1 \le j < n$, and $1 \le l \le \min(j, k)$, where i denotes the vertex the counter is associated to, j denotes the row, and l denotes the

[1] If $k > \lfloor n/2 \rfloor$, it is more efficient to count the number of inactive outgoing arcs up to $n-k$.

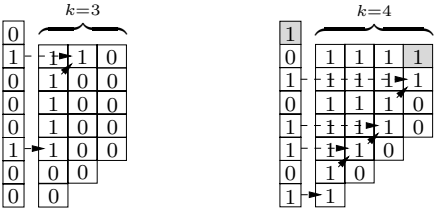


Fig. 3. Sequential unary counter that counts to 2 within its bound $k = 3$ (left) and that counts to 4 and exceeds its bound $k = 4$ as indicated by the shaded boxes (right)

column. The first kind of clauses added to encode the counter behavior are of the form $ctr_{i,j-1,l} \rightarrow ctr_{i,j,l}$ for $1 \leq i \leq n$, $1 < j < n$, and $1 \leq l \leq \min(j, k)$. These clauses ensure that the counter is never decremented, i.e., if some counter variable is assigned true, then all succeeding variables in the same column must be assigned true. Next we add clauses to enforce the actual counting. This is done by clauses of the form

$$arc_{i,j} \rightarrow ctr_{i,j,1}$$

for $1 \leq i \leq n$ and $1 \leq j < n$, which ensure that the counter is at least one if there is an active arc in the current row, and clauses of the form

$$(arc_{i,j} \wedge ctr_{i,j-1,l-1}) \rightarrow ctr_{i,j,l}$$

for $1 \leq i \leq n$, $1 < j < n$, and $1 < l \leq \min(j, k)$, which ensure that the counter is incremented from value $l - 1$ in the previous row to value l in the current row if there is an active arc in the current row. The effect of these clauses is indicated by the arrows in Figure 3. Finally, we enforce a conflict if any counter exceeds k by adding clauses of the form $\neg(arc_{i,j} \wedge ctr_{i,j-1,k})$ for $1 \leq i \leq n$ and $k < j < n$. Such a conflict is indicated by the shaded boxes in the right scenario of Figure 3.

Table 1 shows for several benchmark graphs from computational biology [7] the improvements of the bounds on the treewidth we have achieved with our encoding. To solve our SAT instances we used MiniSAT on a quad-core Intel Xeon CPU with 2.33GHz and 24GB RAM. The timeout for a single run was one hour. We observed that for graphs whose treewidth is known, the SAT solver could find a solution very fast if k was set to the treewidth and it took very long if k was set just below the treewidth. Thus, it can be seen as an indicator that there is no tree decomposition of width k if, for moderately sized graphs, the SAT solver does not find a solution within the timeout.

Table 1. Previous and new lower and upper bounds on the treewidth of four benchmark graphs. The last column shows the time to decide the obtained SAT instance with the new upper bound.

Graph	$\|V\|$	$\|E\|$	Prev LB [7]	Prev UB [7]	New LB	New UB	Time
1c75	69	683	28	30	28	29	15.9s
1dj7	73	743	26	27	26	26	134.0s
1dp7	76	769	25	27	25	26	118.3s
1en2	69	463	16	17	16	16	180.7s

4 Conclusion

Several theoretical and experimental attempts have been made in the past to attack the problem of determining the treewidth of graphs, but to the best of our knowledge this is the first time a SAT solver has been used for this. Although the graphs we used are relatively small and the improvements are not dramatic, we consider the presented approach as a first step towards computing the treewidth of graphs by SAT solvers. One of the main problems to be considered in future work is the size of the resulting SAT instances: Improved encodings that exploit the structure of a given graph may be able to keep the size of the instances small. Further work could be to identify redundant clauses that accelerate the SAT solving process and to adapt the encoding for related graph concepts like hypertree decomposition.

References

1. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM Journal on Algebraic and Discrete Methods 8(2), 277–284 (1987)
2. Bachoore, E.H., Bodlaender, H.L.: New upper bound heuristics for treewidth. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 216–227. Springer, Heidelberg (2005)
3. Bachoore, E.H., Bodlaender, H.L.: A branch and bound algorithm for exact, upper, and lower bounds on treewidth. Technical Report UU-CS-2006-012, Department of Information and Computing Sciences, Utrecht University (2006)
4. Bodlaender, H.L.: Discovering treewidth. In: Vojtáš, P., Bieliková, M., Charron-Bost, B., Sýkora, O. (eds.) SOFSEM 2005. LNCS, vol. 3381, pp. 1–16. Springer, Heidelberg (2005)
5. Bodlaender, H.L., Grigoriev, A., Arie, M.C., Koster, A.: Treewidth lower bounds with brambles. Algorithmica 51, 81–98 (2008)
6. Bodlaender, H.L., Koster, A.M.C.A., Wolle, T.: Contraction and treewidth lower bounds. Journal of Graph Algorithms and Applications (JGAA) 10(1), 5–49 (2006)
7. van den Broek, J.-W., Bodlaender, H.L.: TreewidthLIB (March 2009), http://people.cs.uu.nl/hansb/treewidthlib/
8. Clautiaux, F., Carlier, J., Moukrim, A., Nègre, S.: New lower and upper bounds for graph treewidth. In: Jansen, K., Margraf, M., Mastrolli, M., Rolim, J.D.P. (eds.) WEA 2003. LNCS, vol. 2647, pp. 70–80. Springer, Heidelberg (2003)
9. Courcelle, B.: Graph rewriting: An algebraic and logic approach. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science. Formal Models and Semantics, vol. B, ch. 5, pp. 193–242. Elsevier, Amsterdam (1990)
10. Dechter, R.: Tractable structures for constraint satisfaction problems. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, ch. 7, pp. 209–244. Elsevier, Amsterdam (2006)
11. Gogate, V., Dechter, R.: A complete anytime algorithm for treewidth. In: Proc. of the 20th Conference on Uncertainty in Artificial Intelligence (UAI 2004). ACM International Conference Proceeding Series, vol. 70, pp. 201–208. AUAI Press (2004)
12. Koster, A.M.C.A., Bodlaender, H.L., van Hoesel, S.P.M.: Treewidth: Computational experiments. Electronic Notes in Discrete Mathematics 8, 54–57 (2001); Extended version available as Technical Report ZIB-Report 01-38, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
13. Robertson, N., Seymour, P.D.: Graph minors II. Algorithmic aspects of tree-width. Journal of Algorithms 7, 309–322 (1986)
14. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)

The Complexity of Reasoning for Fragments of Default Logic*

Olaf Beyersdorff, Arne Meier, Michael Thomas, and Heribert Vollmer

Institut für Theoretische Informatik, Gottfried Wilhelm Leibniz Universität
Appelstr. 4, 30167 Hannover, Germany
{beyersdorff,meier,thomas,vollmer}@thi.uni-hannover.de

Abstract. Default logic was introduced by Reiter in 1980. In 1992, Gottlob classified the complexity of the extension existence problem for propositional default logic as Σ_2^{p}-complete, and the complexity of the credulous and skeptical reasoning problem as Σ_2^{p}-complete, resp. Π_2^{p}-complete. Additionally, he investigated restrictions on the default rules, i. e., semi-normal default rules. Selman made in 1992 a similar approach with disjunction-free and unary default rules. In this paper we systematically restrict the set of allowed propositional connectives. We give a complete complexity classification for all sets of Boolean functions in the meaning of Post's lattice for all three common decision problems for propositional default logic. We show that the complexity is a trichotomy (Σ_2^{p}-, NP-complete, trivial) for the extension existence problem, whereas for the credulous and sceptical reasoning problem we get a finer classification down to NL-complete cases.

1 Introduction

When formal specifications are to be verified against real-world situations, one has to overcome the *qualification problem* that denotes the impossibility of listing all conditions required to decide compliance with the specification. To overcome this problem, McCarthy proposed the introduction of "common-sense" into formal logic [McC80]. Among the formalisms developed since then, Reiter's default logic is one of the best known and most successful formalisms for modeling of common-sense reasoning. Default logic extends the usual logical (first-order or propositional) derivations by patterns for default assumptions. These are of the form "in the absence of contrary information, assume . . .". Reiter argued that his logic is an adequate formalization of the human reasoning under the *closed world assumption*. In fact, today default logic is widely used in artificial intelligence and computational logic.

What makes default logic computationally presumably harder than propositional or first-order logic is the fact that the semantics (i. e., the set of consequences) of a given set of premises is defined in terms of a fixed-point equation. The different fixed points (known as *extensions* or *expansions*) correspond to different possible sets of knowledge of an agent, based on the given premises.

* Supported in part by DFG grant VO 630/6-1.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 51–64, 2009.

In a seminal paper from 1992, Georg Gottlob formally defined three important decision problems for default logic:

1. Given a set of premises, decide whether it has an extension at all.
2. Given a set of premises and a formula, decide whether the formula occurs in at least one extension (so called *brave* or *credulous reasoning*).
3. Given a set of premises and a formula, decide whether the formula occurs in all extensions (*cautious* or *sceptical reasoning*).

While in the case of first-order default logic, all these computational tasks are undecidable, Gottlob proved that for *propositional default logic*, the first and second are complete for the class Σ_2^{p}, the second level of the polynomial hierarchy, while the third is complete for the class Π_2^{p}.

In the past, various semantic and syntactic restrictions have been proposed in order to identify computationally easier or even tractable fragments (see, e. g., [Sti90, KS91, BEZ02]). This is the starting point of the present paper. We propose a systematic study of fragments of default logic defined by restricting the set of allowed propositional connectives. For instance, if we look at the fragment where we forbid negation and allow only conjunction and disjunction, the *monotone fragment* of default logic, we show that while the first problem is trivial (there always is an extension, in fact a unique one), the second and third problem become coNP-complete. In this paper we look at all possible sets B of propositional connectives and study the three decision problems defined by Gottlob when all involved formulae contain only connectives from B. The computational complexity of the problems then, of course, becomes a function of B. We will see that *Post's lattice* of all closed classes of Boolean functions is the right way to study all such sets B. Depending on the location of B in this lattice, we completely classify the complexity of all three reasoning tasks, see Figs. 1 and 2. We will show that, depending on the set B of occurring connectives, the problem to determine the existence of an extension is either Σ_2^{p}-complete, NP-complete, or trivial, while the other two problems are complete in one of the classes Σ_2^{p} (or Π_2^{p}), NP, coNP, P or NL (under first-order reductions).

The motivation behind our approach lies in the hope that identifying fragments of default logic with simpler reasoning procedures may help us to understand the sources of hardness for the full problem and to locate the boundary between hard and easy fragments. Especially the improved algorithms can help doing better approaches in solving the studied problems quite more efficiently.

This paper is organized as follows. After some preliminary remarks in Sect. 2, we introduce Boolean clones in Sect. 3. At this place we also provide a full classification of the complexity of logical implications for fragments of propositional logic, as this classification will serve as a central tool for subsequent sections. In Sect. 4, we start to investigate propositional default logic. Section 5 then presents our main results on the complexity of the decision problems for default logic. Finally, in Sect. 6 we conclude with a summary and a discussion.

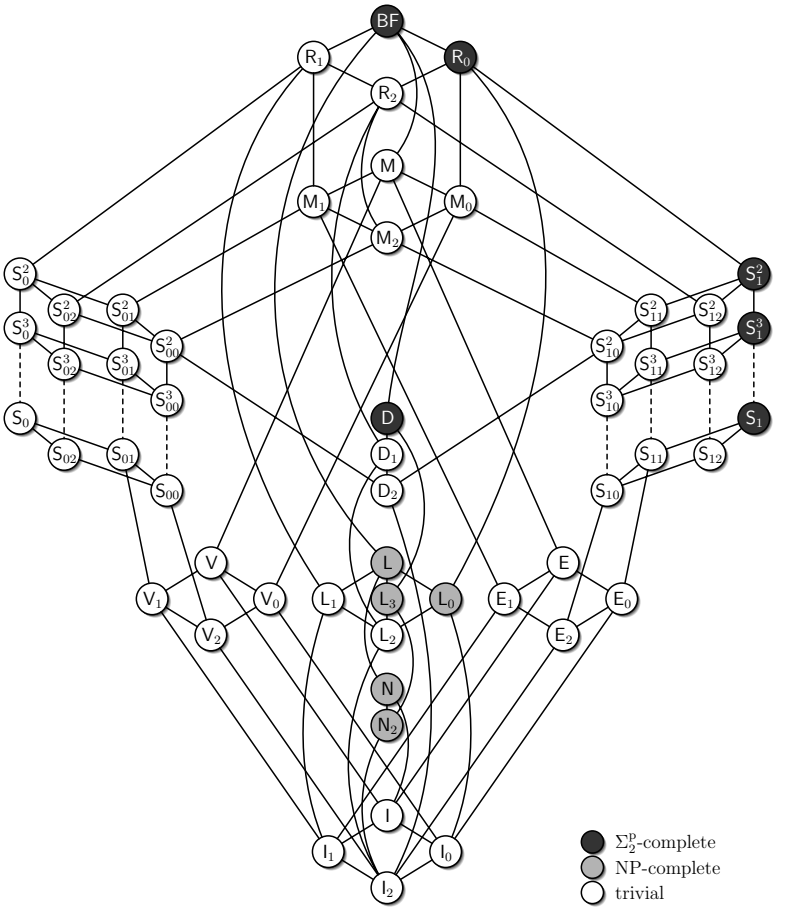


Fig. 1. Post's lattice. Colors indicate the complexity of EXT(B), the Extension Existence Problem for B-formulae.

2 Preliminaries

In this paper we make use of standard notions of complexity theory. The arising complexity degrees encompass the classes NL, P, NP, coNP, Σ_2^{p} and Π_2^{p} (cf. [Pap94] for background information). For the hardness results *constant-depth* reductions are used, defined as follows: A language A is *constant-depth reducible* to a language B ($A \leq_{\mathrm{cd}} B$) if there exists a logtime-uniform AC^0-circuit family $\{C_n\}_{n\geq 0}$ with unbounded fan-in $\{\wedge, \vee, \neg\}$-gates and oracle gates for B such that for all x, $C_{|x|}(x) = 1$ iff $x \in A$ (cf. [Vol99]).

We also assume familiarity with propositional logic. The set of all propositional formulae is denoted by $\mathcal{L}$. For $A \subseteq \mathcal{L}$ and $\varphi \in \mathcal{L}$, we write $A \models \varphi$ iff all assignments satisfying all formulae in A also satisfy φ. By $\mathrm{Th}(A)$ we denote the set of all consequences of A, i. e., $\mathrm{Th}(A) = \{\varphi \mid A \models \varphi\}$. For a literal l and a

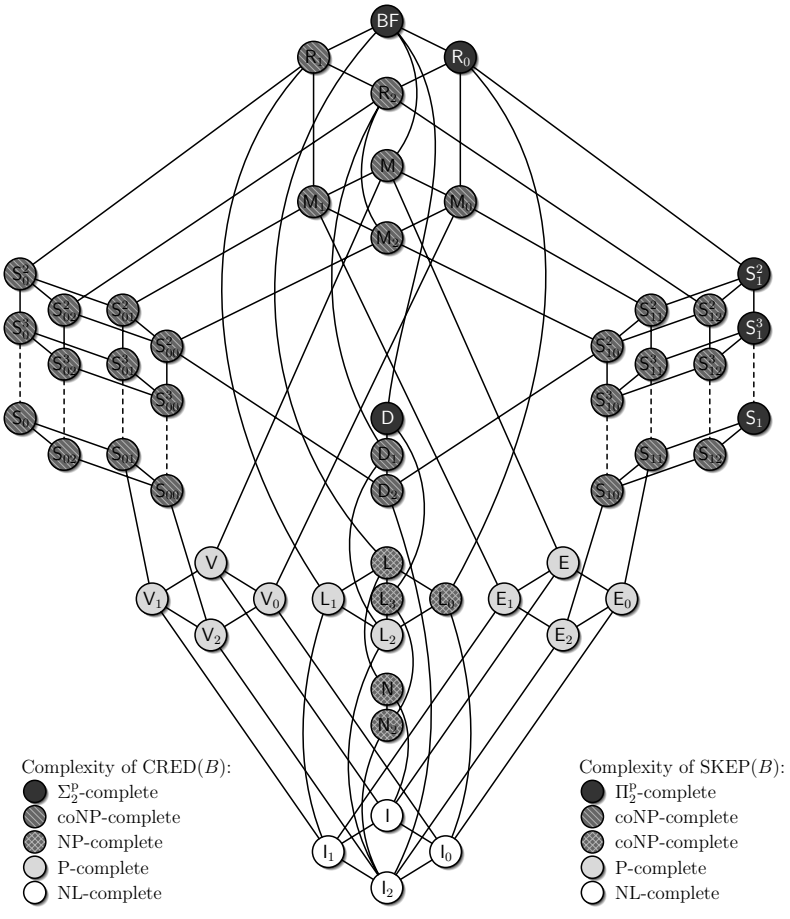


Fig. 2. Post's lattice. Colors indicate the complexity of CRED(B) and SKEP(B), the Credulous and Skeptical Reasoning Problem for B-formulae.

variable x, we define the meta-language expression $\sim l$ as $\sim l := x$ if $l = \neg x$ and $\sim l := \neg x$ if $l = x$. For a formula φ, let $\varphi_{[\alpha/\beta]}$ denote φ with all occurrences of α replaced by β, and let $A_{[\alpha/\beta]} := \{\varphi_{[\alpha/\beta]} \mid \varphi \in A\}$ for $A \subseteq \mathcal{L}$.

3 Boolean Clones and the Complexity of the Implication Problem

A propositional formula using only connectives from a finite set B of Boolean functions is called a B-formula. The set of all B-formulae is denoted by $\mathcal{L}(B)$. In order to cope with the infinitely many finite sets B of Boolean functions, we require some algebraic tools to classify the complexity of the infinitely many arising reasoning problems. A *clone* is a set B of Boolean functions that is closed

Table 1. A list of Boolean clones with definitions and bases

Name	Definition	Base
BF	All Boolean functions	$\{\wedge, \neg\}$
R_0	$\{f : f$ is 0-reproducing$\}$	$\{\wedge, \not\rightarrow\}$
R_1	$\{f : f$ is 1-reproducing$\}$	$\{\vee, \rightarrow\}$
M	$\{f : f$ is monotone$\}$	$\{\vee, \wedge, 0, 1\}$
S_0	$\{f : f$ is 0-separating$\}$	$\{\rightarrow\}$
S_1	$\{f : f$ is 1-separating$\}$	$\{\not\rightarrow\}$
S_{00}	$\mathsf{S}_0 \cap \mathsf{R}_0 \cap \mathsf{R}_1 \cap \mathsf{M}$	$\{x \vee (y \wedge z)\}$
S_{10}	$\mathsf{S}_1 \cap \mathsf{R}_0 \cap \mathsf{R}_1 \cap \mathsf{M}$	$\{x \wedge (y \vee z)\}$
D	$\{f : f$ is self-dual$\}$	$\{(x \wedge \overline{y}) \vee (x \wedge \overline{z}) \vee (\overline{y} \wedge \overline{z})\}$
D_2	$\mathsf{D} \cap \mathsf{M}$	$\{(x \wedge y) \vee (y \wedge z) \vee (x \wedge z)\}$
L	$\{f : f$ is linear$\}$	$\{\oplus, 1\}$
L_0	$\mathsf{L} \cap \mathsf{R}_0$	$\{\oplus\}$
L_1	$\mathsf{L} \cap \mathsf{R}_1$	$\{\equiv\}$
L_2	$\mathsf{L} \cap \mathsf{R}_0 \cap \mathsf{R}_1$	$\{x \oplus y \oplus z\}$
L_3	$\mathsf{L} \cap \mathsf{D}$	$\{x \oplus y \oplus z, \neg\}$
V	$\{f : f \equiv c_0 \vee \bigvee_{i=1}^{n} c_i x_i$ where the c_is are constant$\}$	$\{\vee, 0, 1\}$
V_2	$[\{\vee\}]$	$\{\vee\}$
E	$\{f : f \equiv c_0 \wedge \bigwedge_{i=1}^{n} c_i x_i$ where the c_is are constant$\}$	$\{\wedge, 0, 1\}$
E_2	$[\{\wedge\}]$	$\{\wedge\}$
N	$\{f : f$ depends on at most one variable$\}$	$\{\neg, 0, 1\}$
N_2	$[\{\neg\}]$	$\{\neg\}$
I	$\{f : f$ is a projection or a constant$\}$	$\{\mathrm{id}, 0, 1\}$
I_2	$[\{id\}]$	$\{\mathrm{id}\}$

under superposition, i. e., B contains all projections and is closed under arbitrary composition. For an arbitrary set B of Boolean functions, we denote by $[B]$ the smallest clone containing B and call B a *base* for $[B]$. In [Pos41] Post classified the lattice of all clones and found a finite base for each clone, see Fig. 1. In order to introduce the clones relevant to this paper, we define the following notions for n-ary Boolean functions f:

- f is *c-reproducing* if $f(c, \ldots, c) = c$, $c \in \{0, 1\}$.
- f is *monotone* if $a_1 \leq b_1, a_2 \leq b_2, \ldots, a_n \leq b_n$ implies $f(a_1, \ldots, a_n) \leq f(b_1, \ldots, b_n)$.
- f is *c-separating* if there exists an $i \in \{1, \ldots, n\}$ such that $f(a_1, \ldots, a_n) = c$ implies $a_i = c$, $c \in \{0, 1\}$.
- f is *self-dual* if $f \equiv \mathrm{dual}(f)$, where $\mathrm{dual}(f)(x_1, \ldots, x_n) = \neg f(\neg x_1, \ldots, \neg x_n)$.
- f is *linear* if $f \equiv x_1 \oplus \cdots \oplus x_n \oplus c$ for a constant $c \in \{0, 1\}$ and variables $x_1, \ldots, x_n$.

The clones relevant to this paper are listed in Table 1. The definition of all Boolean clones can be found, e. g., in [BCRV03].

For a finite set B of Boolean functions, we define the *Implication Problem* for B-formulae $\mathrm{IMP}(B)$ as the following computational task: given a set A of B-formulae and a B-formula φ, decide whether $A \models \varphi$ holds. The complexity of

the implication problem is classified in [BMTV08]. The results relevant to this paper are summarized in the following theorem.

Theorem 3.1 ([BMTV08, Theorem 4.1]). *Let B be a finite set of Boolean functions. Then* $\mathrm{IMP}(B)$ *is*

1. coNP-*complete if* $\mathsf{S}_{00} \subseteq [B]$, $\mathsf{S}_{10} \subseteq [B]$ *or* $\mathsf{D}_2 \subseteq [B]$, *and*
2. *in* P *for all other cases.*

A proof of Theorem 3.1 will be included in the full version of this paper.

4 Default Logic

Fix some finite set B of Boolean functions and let α, β, γ be propositional B-formulae. A *B-default (rule)* is an expression $d = \frac{\alpha:\beta}{\gamma}$; α is called *prerequisite*, β is called *justification* and γ is called *consequent* of d. A *B-default theory* is a pair $\langle W, D\rangle$, where W is a set of propositional B-formulae and D is a set of B-default rules. Henceforth we will omit the prefix "B-" if $B = \mathsf{BF}$ or the meaning is clear from the context.

For a given default theory $\langle W, D\rangle$ and a set of formulae E, let $\Gamma(E)$ be the smallest set of formulae such that

1. $W \subseteq \Gamma(E)$,
2. $\Gamma(E)$ is closed under deduction, i. e., $\Gamma(E) = \mathrm{Th}(\Gamma(E))$, and
3. for all defaults $\frac{\alpha:\beta}{\gamma} \in D$ with $\alpha \in \Gamma(E)$ and $\neg\beta \notin E$, it holds that $\gamma \in \Gamma(E)$.

A *(stable) extension* of $\langle W, D\rangle$ is a fixpoint of Γ, i. e., a set E such that $E = \Gamma(E)$.

The following theorem by Reiter provides an alternative characterization of extensions:

Theorem 4.1 ([Rei80]). *Let $\langle W, D\rangle$ be a default theory and E be a set of formulae.*

1. *Let $E_0 = W$ and $E_{i+1} = \mathrm{Th}(E_i) \cup \left\{\gamma \;\middle|\; \frac{\alpha:\beta}{\gamma} \in D, \alpha \in E_i \text{ and } \neg\beta \notin E\right\}$. Then E is a stable extension of $\langle W, D\rangle$ iff $E = \bigcup_{i\in\mathbb{N}} E_i$.*
2. *Let $G = \left\{\frac{\alpha:\beta}{\gamma} \in D \;\middle|\; \alpha \in E \text{ and } \neg\beta \notin E\right\}$. If E is a stable extension of $\langle W, D\rangle$, then*
$$E = \mathrm{Th}(W \cup \{\gamma \mid \tfrac{\alpha:\beta}{\gamma} \in G\}).$$
In this case, G is also called the set of generating defaults *for E.*

Note that stable extensions need not be consistent. However, the following proposition shows that this only occurs if the set W is inconsistent already.

Proposition 4.2 ([MT93, Corollary 3.60]). *Let $\langle W, D\rangle$ be a default theory. Then $\mathcal{L}$ is a stable extension of $\langle W, D\rangle$ iff W is inconsistent.*

As a consequence we obtain:

Corollary 4.3. *Let $\langle W, D\rangle$ be a default theory.*

- *If W is consistent, then every stable extension of $\langle W, D\rangle$ is consistent.*
- *If W is inconsistent, then $\langle W, D\rangle$ has a stable extension.*

The main reasoning tasks in nonmonotonic logics give rise to the following three decision problems:

1. the *Extension Existence Problem* EXT(B)
   Instance: a B-default theory $\langle W, D\rangle$
   Question: Does $\langle W, D\rangle$ have a stable extension?
2. the *Credulous Reasoning Problem* CRED(B)
   Instance: a B-formula φ and a B-default theory $\langle W, D\rangle$
   Question: Is there a stable extension of $\langle W, D\rangle$ that includes φ?
3. the *Skeptical Reasoning Problem* SKEP(B)
   Instance: a B-formula φ and a B-default theory $\langle W, D\rangle$
   Question: Does every stable extension of $\langle W, D\rangle$ include φ?

The next theorem follows from [Got92] and states the complexity of the above decision problems for the general case $[B] = \mathsf{BF}$.

Theorem 4.4 ([Got92]). *Let B be a finite set of Boolean functions such that $[B] = \mathsf{BF}$. Then* EXT(B) *and* CRED(B) *are Σ_2^{p}-complete, whereas* SKEP(B) *is Π_2^{p}-complete.*

5 The Complexity of Default Reasoning

In this section we will classify the complexity of the three problems EXT(B), CRED(B), and SKEP(B) for all choices of Boolean connectives B. We start with some preparations which will substantially reduce the number of cases we have to consider.

Lemma 5.1. *Let* P *be any of the problems* EXT, CRED, *or* SKEP. *Then for each finite set B of Boolean functions,* $\mathrm{P}(B) \equiv_{\mathrm{cd}} \mathrm{P}(B \cup \{1\})$.

Proof. The reductions $\mathrm{P}(B) \leq_{\mathrm{cd}} \mathrm{P}(B \cup \{1\})$ are obvious. For the converse reductions, we will essentially substitute the constant 1 by a new variable t that is forced to be true (this trick goes already back to Lewis [Lew79]). For EXT, the reduction is given by $\langle W, D\rangle \mapsto \langle W', D'\rangle$, where $W' = W_{[1/t]} \cup \{t\}$, $D' = D_{[1/t]}$, and t is a new variable not occurring in $\langle W, D\rangle$. If $\langle W', D'\rangle$ possesses a stable extension E', then $t \in E'$. Hence, $E'_{[t/1]}$ is a stable extension of $\langle W, D\rangle$. On the other hand, if E is a stable extension of $\langle W, D\rangle$, then $\mathrm{Th}(E_{[1/t]} \cup \{t\}) = E_{[1/t]}$ is a stable extension of $\langle W', D'\rangle$. Therefore, each extension E of $\langle W, D\rangle$ corresponds to the extension $E_{[1/t]}$ of $\langle W', D'\rangle$, and vice versa.

For the problems CRED and SKEP, it suffices to note that the above reduction $\langle W, D\rangle \mapsto \langle W', D'\rangle$ has the additional property that for each formula φ and each extension E of $\langle W, D\rangle$, $\varphi \in E$ iff $\varphi_{[1/t]} \in E_{[1/t]}$. □

The next lemma shows that, quite often, B-default theories have unique extensions.

Lemma 5.2. *Let B be a finite set of Boolean functions such that $[B] \subseteq \mathsf{R}_1$ or $[B] \subseteq \mathsf{M}$. Let $\langle W, D\rangle$ be a B-default theory with finite D. Then $\langle W, D\rangle$ has a unique stable extension.*

Proof. For $[B] \subseteq \mathsf{R}_1$, every premise, justification and consequent is 1-reproducing. As all consequences of 1-reproducing functions are again 1-reproducing and the negation of a 1-reproducing function is not 1-reproducing, the justifications in D become irrelevant. Hence the characterization of stable extensions from the first item in Theorem 4.1 simplifies to the following iterative construction: $E_0 = W$ and $E_{i+1} = \mathrm{Th}(E_i) \cup \left\{\gamma \mid \frac{\alpha:\beta}{\gamma} \in D, \alpha \in E_i\right\}$. As D is finite, this construction terminates after finitely many steps, i. e., $E_k = E_{k+1}$ for some $k \geq 0$. Then $E = \bigcup_{i \leq k} E_i$ is the unique stable extension of $\langle W, D\rangle$. For a similar result confer [BO02, Theorem 4.6].

For $[B] \subseteq \mathsf{M}$, every formula is either 1-reproducing or equivalent to 0. As rules with justification equivalent to 0 are never applicable, each B-default theory $\langle W, D\rangle$ with finite D has a unique stable extension by the same argument as above. □

As an immediate corollary, the credulous and the sceptical reasoning problem are equivalent for the above choices of the underlying connectives.

Corollary 5.3. *Let B be a finite set of Boolean functions such that $[B] \subseteq \mathsf{R}_1$ or $[B] \subseteq \mathsf{M}$. Then* $\mathrm{CRED}(B) \equiv_{\mathrm{cd}} \mathrm{SKEP}(B)$.

5.1 The Extension Existence Problem

Now we are ready to classify the complexity of EXT. The next theorem shows that this is a trichotomy: the Σ_2^{p}-completeness of the general case [Got92] is inherited by all clones above S_1 and D, for a number of clones the complexity of EXT reduces to NP-completeness, and, due to Lemma 5.2, for the majority of cases the problem becomes trivial.

Theorem 5.4. *Let B be a finite set of Boolean functions. Then* $\mathrm{EXT}(B)$ *is*

1. Σ_2^{p}*-complete if* $\mathsf{S}_1 \subseteq [B] \subseteq \mathsf{BF}$ *or* $\mathsf{D} \subseteq [B] \subseteq \mathsf{BF}$,
2. NP*-complete if* $[B] \in \{\mathsf{N}, \mathsf{N}_2, \mathsf{L}, \mathsf{L}_0, \mathsf{L}_3\}$, *and*
3. *trivial in all other cases (i. e., if* $[B] \subseteq \mathsf{R}_1$ *or* $[B] \subseteq \mathsf{M}$*).*

Proof. For $\mathsf{S}_1 \subseteq [B] \subseteq \mathsf{BF}$ or $[B] = \mathsf{D}$, observe that in both cases $\mathsf{BF} = [B \cup \{1\}]$. Claim 1 then follows from Theorem 4.4 and Lemma 5.1.

For the second claim, it suffices to prove membership in NP for $\mathrm{EXT}(B)$ for every finite $[B] \subseteq \mathsf{L}$ and NP-hardness for $\mathrm{EXT}(B)$ for every finite B with $\mathsf{N} \subseteq [B]$. The remaining cases $[B] \in \{\mathsf{N}_2, \mathsf{L}_0, \mathsf{L}_3\}$ all follow from Lemma 5.1, because $[\mathsf{N}_2 \cup \{1\}] = \mathsf{N}$, $[\mathsf{L}_0 \cup \{1\}] = \mathsf{L}$, and $[\mathsf{L}_3 \cup \{1\}] = \mathsf{L}$.

We start by showing $\mathrm{EXT}(\mathsf{L}) \in \mathrm{NP}$. Given a default theory $\langle W, D\rangle$, we first guess a set $G \subseteq D$ which will serve as the set of generating defaults for a stable

extension. Let $G' = W \cup \{\gamma \mid \frac{\alpha:\beta}{\gamma} \in G\}$. We use Theorem 4.1 to verify whether $\mathrm{Th}(G')$ is indeed a stable extension of $\langle W, D\rangle$. For this we inductively compute generators G_i for the sets E_i from Theorem 4.1, until eventually $E_i = E_{i+1}$ (note, that because D is finite, this always occurs). We start by setting $G_0 = W$. Given G_i, we check for each rule $\frac{\alpha:\beta}{\gamma} \in D$, whether $G_i \models \alpha$ and $G' \not\models \neg\beta$ (as all formulae belong to $\mathcal{L}(B)$, this is possible by Theorem 3.1). If so, then γ is put into G_{i+1}. If this process terminates, i. e., if $G_i = G_{i+1}$, then we check whether $G' = G_i$. By Theorem 4.1, this test is positive iff G generates a stable extension of $\langle W, D\rangle$.

To show NP-hardness of EXT(B) for $\mathsf{N} \subseteq [B]$, we will $\leq_{\mathrm{cd}}$-reduce 3SAT to EXT(B). Let $\varphi = \bigwedge_{i=1}^{n}(l_{i1} \vee l_{i2} \vee l_{i3})$ and l_{ij} be literals over propositions $\{x_1, \ldots, x_m\}$ for $1 \leq i \leq n$, $1 \leq j \leq 3$. We transform φ to the B-default theory $\langle W, D_\varphi\rangle$, where $W := \emptyset$ and

$$D_\varphi := \left\{ \frac{1 : x_i}{x_i} \;\middle|\; 1 \leq i \leq m \right\} \cup \left\{ \frac{1 : \neg x_i}{\neg x_i} \;\middle|\; 1 \leq i \leq m \right\} \cup \left\{ \frac{\sim l_{i\pi(1)} : \sim l_{i\pi(2)}}{l_{i\pi(3)}} \;\middle|\; 1 \leq i \leq n, \pi \text{ is a permutation of } \{1,2,3\} \right\} .$$

To prove the correctness of the reduction, first assume φ to be satisfiable. For each satisfying assignment $\sigma : \{x_1, \ldots, x_m\} \to \{0, 1\}$ for φ, we claim that

$$E := \mathrm{Th}(\{x_i \mid \sigma(x_i) = 1\} \cup \{\neg x_i \mid \sigma(x_i) = 0\})$$

is a stable extension of $\langle W, D_\varphi\rangle$. We will verify this claim with the help of the first part of Theorem 4.1. Starting with $E_0 = \emptyset$, we already get $E_1 = E$ by the default rules $\frac{1:x_i}{x_i}$ and $\frac{1:\neg x_i}{\neg x_i}$ in D_φ. As σ is a satisfying assignment for φ, each consequent of a default rule in D_φ is already in E. Hence $E_2 = E_1$ and therefore $E = \bigcup_{i\in\mathbb{N}} E_i$ is a stable extension of $\langle W, D_\varphi\rangle$.

Conversely, assume that E is a stable extension of $\langle W, D_\varphi\rangle$. Because of the default rules $\frac{1:x_i}{x_i}$ and $\frac{1:\neg x_i}{\neg x_i}$, we either get $x_i \in E$ or $\neg x_i \in E$ for all $i = 1, \ldots, m$. The rules of the type $\frac{\sim l_{i1}:\sim l_{i2}}{l_{i3}}$ ensure that E contains at least one literal from each clause $l_{i1} \vee l_{i2} \vee l_{i3}$ in φ. As E is deductively closed, E contains φ. By Corollary 4.3, the extension E is consistent, and therefore φ is satisfiable.

Finally, the third item of the theorem directly follows from Lemma 5.2. □

5.2 The Credulous and the Sceptical Reasoning Problem

Now we will analyse the credulous and the sceptical reasoning problem. For these problems, there are two sources for the complexity. On the one hand, we need to determine a candidate for a stable extension. On the other hand, we have to verify that this candidate is indeed a finite characterization of some stable extension — a task that requires to test for formula implication. Whence the Σ_2^{p}-completeness of CRED(B) and the Π_2^{p}-completeness of SKEP(B) if $[B] = \mathsf{BF}$ derives. Depending on the Boolean connectives allowed, one or both tasks can be performed in polynomial time. We obtain coNP-completeness for clones that

guarantee the existence of a stable extension but whose implication problem remains coNP-complete. Conversely, if the implication problem becomes easy, but determining an extension candidates is hard, then CRED(B) is NP-complete, while SKEP(B) has to test for all extensions and is coNP-complete. This is the case for the clones $[B] \in \{\mathsf{N}, \mathsf{N}_2, \mathsf{L}, \mathsf{L}_0, \mathsf{L}_3\}$. Finally, for clones B that allow for solving both tasks in polynomial time, CRED(B) and SKEP(B) are in P. The complete classification of CRED(B) is given in the following theorem.

Theorem 5.5. *Let B be a finite set of Boolean functions. Then* CRED(B) *is*

1. Σ_2^{p}*-complete if* $\mathsf{S}_1 \subseteq [B] \subseteq \mathsf{BF}$ *or* $\mathsf{D} \subseteq [B] \subseteq \mathsf{BF}$,
2. coNP*-complete if* $X \subseteq [B] \subseteq Y$, *where* $X \in \{\mathsf{S}_{00}, \mathsf{S}_{10}, \mathsf{D}_2\}$ *and* $Y \in \{\mathsf{R}_1, \mathsf{M}\}$,
3. NP*-complete if* $[B] \in \{\mathsf{N}, \mathsf{N}_2, \mathsf{L}, \mathsf{L}_0, \mathsf{L}_3\}$,
4. P*-complete if* $\mathsf{V}_2 \subseteq [B] \subseteq \mathsf{V}$, $\mathsf{E}_2 \subseteq [B] \subseteq \mathsf{E}$ *or* $[B] \in \{\mathsf{L}_1, \mathsf{L}_2\}$, *and*
5. NL*-complete if* $\mathsf{I}_2 \subseteq [B] \subseteq \mathsf{I}$.

The proof of Theorem 5.5 follows from the upper and lower bounds given in the Propositions 5.6 and 5.7 below.

Proposition 5.6. *Let B be a finite set of Boolean functions. Then* CRED(B) *is contained*

1. *in* Σ_2^{p} *if* $\mathsf{S}_1 \subseteq [B] \subseteq \mathsf{BF}$ *or* $\mathsf{D} \subseteq [B] \subseteq \mathsf{BF}$,
2. *in* coNP *if* $[B] \subseteq \mathsf{R}_1$ *or* $[B] \subseteq \mathsf{M}$,
3. *in* NP *if* $[B] \subseteq \mathsf{L}$,
4. *in* P *if* $[B] \subseteq \mathsf{V}$, $[B] \subseteq \mathsf{E}$ *or* $[B] \subseteq \mathsf{L}_1$, *and*
5. *in* NL *if* $[B] \subseteq \mathsf{I}$.

Proof. Part 1 follows from Theorem 4.4 and Lemma 5.1. For $[B] \subseteq \mathsf{R}_1$, let $\langle W, D\rangle$ be an R_1-default theory and $\varphi \in \mathcal{L}(\mathsf{R}_1)$. As for every default rule $\frac{\alpha:\beta}{\gamma} \in D$ we can never derive $\neg\beta$ (as $\neg\beta$ is not 1-reproducing), the justifications β are irrelevant for computing a stable extension. Thence, using the characterization in the first part of Theorem 4.1, we can iteratively compute the applicable defaults and eventually check whether φ is implied by W and those generating defaults. Algorithm 1 implements these steps on a deterministic Turing machine using a coNP-oracle to test for implication of B-formulae. Clearly, Algorithm 1 terminates after a polynomial number of steps. Moreover, Algorithm 1 is a monotone $\leq_{\mathrm{T}}^{\mathrm{p}}$-reduction from CRED($B$) to IMP($B$),

Algorithm 1. Determine existence of a stable extension of $\langle W, D\rangle$ containing φ

Require: $\langle W, D\rangle$, φ
1: $G_{\mathrm{new}} \leftarrow W$
2: **repeat**
3: $\quad G_{\mathrm{old}} \leftarrow G_{\mathrm{new}}$
4: $\quad$ **for all** $\frac{\alpha:\beta}{\gamma} \in D$ **do**
5: $\quad\quad$ **if** $G_{\mathrm{old}} \models \alpha$ **then**
6: $\quad\quad\quad G_{\mathrm{new}} \leftarrow G_{\mathrm{new}} \cup \{\gamma\}$
7: $\quad\quad$ **end if**
8: $\quad$ **end for**
9: **until** $G_{\mathrm{new}} = G_{\mathrm{old}}$
10: **if** $G_{\mathrm{new}} \models \varphi$ **then**
11: $\quad$ **return true**
12: **else**
13: $\quad$ **return false**
14: **end if**

in the sense that for any deterministic oracle Turing machine M that executes Algorithm 1, $A \subseteq B$ implies $L(M, A) \subseteq L(M, B)$, where $L(M, X)$ is the language recognized by M with oracle X. As coNP is closed under monotone $\leq_{\mathrm{T}}^{\mathrm{p}}$-reductions [Sel82], $\mathrm{CRED}(B) \in \mathrm{coNP}$.

For $[B] \subseteq \mathsf{M}$, Algorithm 1 can be easily adopted, because we are restricted to 1-reproducing functions and the constant 0. Thus, before executing Algorithm 1, we just delete all rules $\frac{\alpha:\beta}{\gamma}$ with $\beta \equiv 0$ from D, as these rules are never applicable.

For $[B] \subseteq \mathsf{L}$, we proceed similarly as in the proof of item 2 in Theorem 5.4. First, we guess a set G of generating defaults and subsequently verify that both $\mathrm{Th}(W \cup \{\gamma \mid \frac{\alpha:\beta}{\gamma} \in G\})$ is a stable extension and that $W \cup \{\gamma \mid \frac{\alpha:\beta}{\gamma} \in G\} \models \varphi$. Using Theorem 3.1, both conditions may be verified in polynomial time.

For $[B] \subseteq \mathsf{V}$, $[B] \subseteq \mathsf{E}$, and $[B] \subseteq \mathsf{L}_1$, we again use Algorithm 1. As for these types of B-formulae, we have an efficient test for implication (Theorem 3.1). Hence, $\mathrm{CRED}(B) \in \mathrm{P}$.

For $[B] \subseteq \mathsf{I}$, we show that $\mathrm{CRED}(B)$ is constant-depth reducible to the graph accessibility problem, GAP, a problem that is $\leq_{\mathrm{cd}}$-complete for NL. Let $\langle W, D\rangle$ be an I-default theory with $D = \{\frac{\alpha_i:\beta_i}{\gamma_i} \mid 1 \leq i \leq k\}$ and let φ be an I-formula. We transform $(\langle W, D\rangle, \varphi)$ to the GAP-instance $(G, \bigwedge_{\psi \in W} \psi, \varphi)$, where $G = (V, E)$ is a directed graph with

$$V := \{\alpha_i \mid 1 \leq i \leq k\} \cup \{\gamma_i \mid 1 \leq i \leq k\} \cup \{\textstyle\bigwedge_{\psi \in W} \psi, \varphi\} \text{ and}$$
$$E := \{(\alpha_i, \gamma_i) \mid 1 \leq i \leq k\} \cup \{(u, v) \in V \times V \mid u \models v\} \ .$$

Then φ is included in the (unique) stable extension of $\langle W, D\rangle$ iff G contains a path from $\bigwedge_{\psi \in W} \psi$ to φ. As implication testing for all $B \subseteq \mathsf{I}$ is possible in AC^0, $\mathrm{CRED}(B) \leq_{\mathrm{cd}} \mathrm{GAP}$. □

We will now establish the lower bounds required to complete the proof of Theorem 5.5.

Proposition 5.7. *Let B be a finite set of Boolean functions. Then* $\mathrm{CRED}(B)$ *is*

1. Σ_2^{p}*-hard if* $\mathsf{S}_1 \subseteq [B]$ *or* $\mathsf{D} \subseteq [B]$,
2. coNP*-hard if* $\mathsf{S}_{00} \subseteq [B]$, $\mathsf{S}_{10} \subseteq [B]$ *or* $\mathsf{D}_2 \subseteq [B]$,
3. NP*-hard if* $\mathsf{N}_2 \subseteq [B]$ *or* $\mathsf{L}_0 \subseteq [B]$,
4. P*-hard if* $\mathsf{V}_2 \subseteq [B]$, $\mathsf{E}_2 \subseteq [B]$ *or* $\mathsf{L}_2 \subseteq [B]$, *and*
5. NL*-hard for all other clones.*

Proof. Part 1 follows from Theorem 4.4 and Lemma 5.1.

For $\mathsf{S}_{00} \subseteq [B]$, $\mathsf{S}_{10} \subseteq [B]$, and $\mathsf{D}_2 \subseteq [B]$, coNP-hardness is established by a $\leq_{\mathrm{cd}}$-reduction from $\mathrm{IMP}(B)$. Let $A \subseteq \mathcal{L}(B)$ and $\varphi \in \mathcal{L}(B)$. Then the default theory $\langle A, \emptyset\rangle$ has the unique stable extension $\mathrm{Th}(A)$, and hence $A \models \varphi$ iff $(\langle A, \emptyset\rangle, \varphi) \in \mathrm{CRED}(B)$. Therefore, $\mathrm{IMP}(B) \leq_{\mathrm{cd}} \mathrm{CRED}(B)$, and the claim follows with Theorem 3.1.

For the third item, it suffices to prove NP-hardness for $\mathsf{N}_2 \subseteq [B]$. For $\mathsf{L}_0 \subseteq [B]$, the claim then follows by Lemma 5.1. For $\mathsf{N}_2 \subseteq [B]$, we obtain NP-hardness of

CRED(B) by adjusting the reduction given in the proof of item 2 of Theorem 5.4. Consider the mapping $\varphi \mapsto (\langle\{\psi\}, D_\varphi\rangle, \psi)$, where D_φ is the set of default rules constructed from φ in Theorem 5.4, and ψ is a satisfiable B-formula such that φ and ψ do not use common variables. By Theorem 5.4, $\varphi \in$ 3SAT iff $\langle\{\psi\}, D_\varphi\rangle$ has a stable extension. As any extension of $\langle\{\psi\}, D_\varphi\rangle$ contains ψ, we obtain 3SAT $\leq_{\text{cd}}$ CRED(B) via the above reduction.

To prove P-hardness for $\mathsf{E}_2 \subseteq [B]$, $\mathsf{V}_2 \subseteq [B]$, and $[B] \in \{\mathsf{L}_1, \mathsf{L}_2\}$, we provide a reduction from the accessibility problem for directed hypergraphs, HGAP. HGAP is P-complete under $\leq_{\text{cd}}$-reductions [SI90]. In directed hypergraphs $H = (V, E)$, hyperedges $e \in E$ consist of a set of source nodes $\text{src}(e) \subseteq V$ and a destination $\text{dest}(e) \in V$. Instances of HGAP contain a directed hypergraph $H = (V, E)$, a set $S \subseteq V$ of source nodes, and a target node $t \in V$. We transform such an instance (H, S, t) to the CRED($\{\wedge\}$)-instance $(\langle W, D\rangle, \varphi)$, where

$$W := \{p_s \mid s \in S\},\ D := \left\{ \frac{\bigwedge_{v\in\text{src}(e)} p_v \ :\ \bigwedge_{v\in\text{src}(e)} p_v}{p_{\text{dest}(e)}} \;\middle|\; e \in E \right\},\ \varphi := p_t$$

with pairwise distinct propositions p_v for $v \in V$. For $\mathsf{V}_2 \subseteq [B]$, we set

$$W := \{\bigvee_{s\notin S} p_s\},\ D := \left\{ \frac{\bigvee_{v\in V\setminus\text{src}(e)} p_v : \bigvee_{v\in V\setminus\text{src}(e)} p_v}{\bigvee_{v\in V\setminus(\text{src}(e)\cup\{\text{dest}(e)\})} p_v} \;\middle|\; e \in E \right\},\ \varphi := \bigvee_{v\in V\setminus\{t\}} p_v.$$

For $[B] \in \{\mathsf{L}_1, \mathsf{L}_2\}$, we again modify the above reduction and map (H, S, t) to the CRED(B)-instance

$$W := \{p_s \mid s \in S\},\ D := \left\{ \frac{\equiv_{v\in\text{src}(e)} p_v \ :\ \equiv_{v\in\text{src}(e)} p_v}{p_{\text{dest}(e)}} \;\middle|\; e \in E \right\},\ \varphi := p_t.$$

The correctness of these reductions is easily verified.

Finally, it remains to show NL-hardness for $\mathsf{I}_2 \subseteq [B]$. We give a $\leq_{\text{cd}}$-reduction from GAP to CRED($\{\text{id}\}$). For a directed graph $G = (V, E)$ and two nodes $s, t \in V$, we transform the GAP-instance (G, s, t) to the CRED(I_2)-instance

$$W := \{p_s\},\ D := \left\{ \frac{p_u : p_u}{p_v} \;\middle|\; (u, v) \in E \right\},\ \varphi := p_t.$$

Clearly, $(G, s, t) \in$ GAP iff φ is contained in all stable extensions of $\langle W, D\rangle$. □

Finally, we will classify the complexity of the sceptical reasoning problem. The analysis is similar to the classification of the credulous reasoning problem (Theorem 5.5).

Theorem 5.8. *Let B be a finite set of Boolean functions. Then* SKEP(B) *is*

1. Π_2^{p}*-complete if* $\mathsf{S}_1 \subseteq [B] \subseteq \mathsf{BF}$ *or* $\mathsf{D} \subseteq [B] \subseteq \mathsf{BF}$,
2. coNP*-complete if* $X \subseteq [B] \subseteq Y$, *where* $X \in \{\mathsf{S}_{00}, \mathsf{S}_{10}, \mathsf{N}_2, \mathsf{L}_0\}$ *and* $Y \in \{\mathsf{R}_1, \mathsf{M}, \mathsf{L}\}$,
3. P*-complete if* $\mathsf{V}_2 \subseteq [B] \subseteq \mathsf{V}$, $\mathsf{E}_2 \subseteq [B] \subseteq \mathsf{E}$ *or* $[B] \in \{\mathsf{L}_1, \mathsf{L}_2\}$, *and*
4. NL*-complete if* $\mathsf{I}_2 \subseteq [B] \subseteq \mathsf{I}$.

Proof. The first part again follows from Theorem 4.4 and Lemma 5.1.

For $[B] \in \{\mathsf{N}, \mathsf{N}_2, \mathsf{L}, \mathsf{L}_0, \mathsf{L}_3\}$, we guess similarly as in Theorem 5.4 a set G of defaults and then verify in the same way whether W and G generate a stable extension E. If not, then we accept. Otherwise, we check if $E \models \varphi$ and answer according to this test. This yields a coNP-algorithm for SKEP(B). Hardness for coNP is achieved by modifying the reduction from Theorem 5.4 (cf. also the proof of Proposition 5.7): map φ to $(\langle \emptyset, D_\varphi \rangle, \psi)$, where D_φ is defined as in the proof of Theorem 5.4, and ψ is a B-formula such that φ and ψ do not share variables. Then $\varphi \notin$ 3SAT iff $\langle \emptyset, D_\varphi \rangle$ does not have a stable extension. The latter is true iff ψ is in all extensions of $\langle \emptyset, D_\varphi \rangle$. Hence $\overline{\text{3SAT}} \leq_{\text{cd}} \text{SKEP}(B)$, establishing the claim.

For all remaining clones B, observe that $[B] \subseteq \mathsf{R}_1$ or $[B] \subseteq \mathsf{M}$. Hence, Corollary 5.3 and Theorem 5.5 imply the claim. □

6 Conclusion

In this paper we provided a complete classification of the complexity of the main reasoning problems for default propositional logic, one of the most common frameworks for nonmonotonic reasoning. The complexity of the extension existence problem shows an interesting similarity to the complexity of the satisfiability problem [Lew79], because in both cases the hardest instances lie above the clone S_1 (with the exception that instances from D are still hard for EXT, but easy for SAT). The complexity of the membership problems, i. e., credulous and skeptical reasoning, rests on two sources: first, whether there exist unique extensions (cf. Lemma 5.2), and second, how hard it is to test for formula implication. For this reason, we also classified the complexity of the implication problem IMP(B).

A different complexity classification of reasoning for default logic has been undertaken in [CHS07]. In that paper, the language of propositional formulas was restricted to so called conjunctive queries, i.e., existentially quantified formulas in conjunctive normal-form with generalized clauses. The complexity of the reasoning tasks was determined depending on the type of clauses that are allowed. We want to remark that though our approach at first sight seems to be more general (since we do not restrict our formulas to CNF), the results in [CHS07] do not follow from the results presented here (and vice versa, our results do not follow from theirs).

In the light of our present contribution, it is interesting to remark that by results of Konolige and Gottlob [Kon88, Got95], propositional default logic and Moore's autoepistemic logic are essentially equivalent. Even more, the translation is efficiently computable. Unfortunately, this translation requires a complete set of Boolean connectives, whence our results do not immediately transfer to autoepistemic logic. It is nevertheless interesting to ask whether the exchange of default rules with the introspective operator L yields further efficiently decidable fragments.

Acknowledgements

We thank Ilka Schnoor for sending us a manuscript with the proof of the NP-hardness of EXT(B) for all B such that $\mathsf{N}_2 \subseteq [B]$. We also acknowledge helpful discussions on various topics of this paper with Peter Lohmann.

References

[BCRV03] Böhler, E., Creignou, N., Reith, S., Vollmer, H.: Playing with Boolean blocks, part I: Post's lattice with applications to complexity theory. SIGACT News 34(4), 38–52 (2003)

[BEZ02] Ben-Eliyahu-Zohary, R.: Yet some more complexity results for default logic. Artificial Intelligence 139(1), 1–20 (2002)

[BMTV08] Beyersdorff, O., Meier, A., Thomas, M., Vollmer, H.: The complexity of propositional implication. ACM Computing Research Repository, arXiv:0811.0959v1 [cs.CC] (2008)

[BO02] Bonatti, P.A., Olivetti, N.: Sequent calculi for propositional nonmonotonic logics. ACM Trans. Comput. Logic 3(2), 226–278 (2002)

[CHS07] Chapdelaine, P., Hermann, M., Schnoor, I.: Complexity of default logic on generalized conjunctive queries. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS, vol. 4483, pp. 58–70. Springer, Heidelberg (2007)

[Got92] Gottlob, G.: Complexity results for nonmonotonic logics. Journal of Logic Computation 2(3), 397–425 (1992)

[Got95] Gottlob, G.: Translating default logic into standard autoepistemic logic. J. ACM 42(4), 711–740 (1995)

[Kon88] Konolige, K.: On the relation between default and autoepistemic logic. Artificial Intelligence 35(3), 343–382 (1988); Erratum: Artificial Intelligence 41(1), 115

[KS91] Kautz, H.A., Selman, B.: Hard problems for simple default logics. Artificial Intelligence 49, 243–279 (1991)

[Lew79] Lewis, H.: Satisfiability problems for propositional calculi. Mathematical Systems Theory 13, 45–53 (1979)

[McC80] McCarthy, J.: Circumscription – a form of non-monotonic reasoning. Artificial Intelligence 13, 27–39 (1980)

[MT93] Marek, V.W., Truszczyński, M.: Nonmonotonic Logic. Artificial Intelligence. Springer, Heidelberg (1993)

[Pap94] Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1994)

[Pos41] Post, E.: The two-valued iterative systems of mathematical logic. Annals of Mathematical Studies 5, 1–122 (1941)

[Rei80] Reiter, R.: A logic for default reasoning. Artificial Intelligence 13, 81–132 (1980)

[Sel82] Selman, A.L.: Reductions on NP and p-selective sets. Theoretical Computer Science 19, 287–304 (1982)

[SI90] Sridhar, R., Iyengar, S.: Efficient parallel algorithms for functional dependency manipulations. In: Proc. 2nd DPDS, pp. 126–137 (1990)

[Sti90] Stillman, J.: It's not my default: The complexity of membership problems in restricted propositional default logics. In: Proc. 8th AAAI, pp. 571–578 (1990)

[Vol99] Vollmer, H.: Introduction to Circuit Complexity. Texts in Theoretical Computer Science. Springer, Heidelberg (1999)

Does Advice Help to Prove Propositional Tautologies?

Olaf Beyersdorff[1] and Sebastian Müller[2,⋆]

[1] Institut für Theoretische Informatik, Leibniz-Universität Hannover, Germany
beyersdorff@thi.uni-hannover.de
[2] Institut für Informatik, Humboldt-Universität zu Berlin, Germany
smueller@informatik.hu-berlin.de

Abstract. One of the starting points of propositional proof complexity is the seminal paper by Cook and Reckhow [6], where they defined propositional proof systems as poly-time computable functions which have all propositional tautologies as their range. Motivated by provability consequences in bounded arithmetic, Cook and Krajíček [5] have recently started the investigation of proof systems which are computed by poly-time functions using advice. While this yields a more powerful model, it is also less directly applicable in practice.

In this note we investigate the question whether the usage of advice in propositional proof systems can be simplified or even eliminated. While in principle, the advice can be very complex, we show that proof systems with logarithmic advice are also computable in poly-time with access to a sparse NP-oracle. In addition, we show that if advice is "not very helpful" for proving tautologies, then there exists an optimal propositional proof system without advice. In our main result, we prove that advice can be transferred from the proof to the formula, leading to an easier computational model. We obtain this result by employing a recent technique by Buhrman and Hitchcock [4].

1 Introduction

Propositional proof complexity studies the question how difficult it is to prove propositional tautologies. In the classical Cook-Reckhow model, proofs are verified in deterministic polynomial time [6]. While this is certainly the most useful setting for practical applications, it is nevertheless interesting to ask if proofs can be shortened when their verification is possible with stronger computational resources. In this direction, Cook and Krajíček [5] have recently initiated the study of proof systems which use advice for the verification of proofs. Their results show that, like in the classical Cook-Reckhow setting, these proof systems enjoy a close connection to theories of bounded arithmetic.

Subsequently, in [2,3] we studied the questions whether there exist polynomially bounded or optimal proof systems with advice. For the first question, one of the major motivations for proof complexity [6], we obtained a complete

⋆ Supported by DFG grant KO 1053/5-2.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 65–72, 2009.

complexity-theoretic characterization [2]. Unlike in the classical model, the second question receives a surprising positive answer: optimal proof systems exist when a small amount of advice is allowed [5,3].

Unfortunately, proof systems with advice do not constitute a feasible model for the verification of proofs in practice, as the non-uniform advice can be very complex (and even non-recursive). In this paper we therefore investigate the question whether the advice can be simplified or even eliminated while still preserving the same upper bounds on the lengths of proofs. Our first result shows that proving propositional tautologies does not require complicated advice: every propositional proof system with up to logarithmic advice is simulated by a propositional proof system computable in polynomial time with access to a sparse NP-oracle. Thus in propositional proof complexity, computation with advice can be replaced by a more realistic computational model.

While this first result holds unconditionally, our next two results explore consequences of a positive or negative answer to our question in the title. Assume first that advice helps to prove tautologies in the sense that proof systems with advice admit non-trivial upper bounds on the lengths of proofs. Then we show that the same upper bound can be achieved in a proof system with a simplified advice model. On the other hand, if the answer is negative in the sense that advice does not help to shorten proofs even for simple tautologies, then we obtain optimal propositional proof systems without advice.

2 Proof Systems with Advice – and without

We start with a general semantic definition of proof systems:

Definition 1. *A* proof system *for a language L is a (possibly partial) surjective function $f : \Sigma^* \to L$. For $L = \mathrm{TAUT}$, f is called a* propositional proof system.

A string w with $f(w) = x$ is called an *f-proof* of x. Proof complexity studies lengths of proofs, so we use the following notion: for a function $t : \mathbb{N} \to \mathbb{N}$, a proof system f for L is *t-bounded* if every $x \in L$ has an f-proof of size $\leq t(|x|)$. If t is a polynomial, then f is called *polynomially bounded*.

In the classical framework of Cook and Reckhow [6], proof systems are additionally required to be computable in polynomial time. Recently, Cook and Krajíček [5] have started to investigate propositional proof systems that are computable in polynomial time with the help of advice. This can be formalized as follows:

Definition 2 ([2]). *For a function $k : \mathbb{N} \to \mathbb{N}$, a proof system f for L is a* proof system with k bits of advice*, if there exist a polynomial-time Turing transducer M, an advice function $h : \mathbb{N} \to \Sigma^*$, and an advice selector function $\ell : \Sigma^* \to 1^*$ such that*

1. *ℓ is computable in polynomial time,*
2. *M computes the proof system f with the help of the advice h, i.e., for all $\pi \in \Sigma^*$, $f(\pi) = M(\pi, h(|\ell(\pi)|))$, and*
3. *for all $n \in \mathbb{N}$, the length of the advice $h(n)$ is bounded by $k(n)$.*

We say that f *uses k bits of input advice* if ℓ has the special form $\ell(\pi) = 1^{|\pi|}$. On the other hand, in case $\ell(\pi) = 1^{|f(\pi)|}$, then f is said to *use k bits of output advice*. The latter notion is only well-defined if we assume that the length of the output $f(\pi)$ (in case $f(\pi)$ is defined) does not depend on the advice. By this definition, proof systems with input advice use non-uniform information depending on the length of the proof, while proof systems with output advice use non-uniform information depending on the length of the proven formula.

In [2] we have shown that every proof system with advice is equivalent to a proof system with the same amount of input advice, whereas output advice seems to yield a strictly less powerful model. Yet, even output advice can be arbitrarily complex and thus computing proofs with advice does not form a feasible model to use in practice. Our first result shows that instead of possibly complex non-uniform information we can also use sparse NP-oracles to achieve the same proof lengths as in proof systems with advice. The qualification "same proof length" is made precise by the notion of simulation [8]: a proof system g *simulates* a proof system f, denoted $f \leq g$, if there is a polynomial p such that for every f-proof π there exists a g-proof π' of size $\leq p(|\pi|)$ with $f(\pi) = g(\pi')$.

Theorem 3

1. *Every propositional proof system with logarithmic advice is simulated by a propositional proof system computable in polynomial time with access to a sparse* NP*-oracle.*
2. *Conversely, every propositional proof system computable in polynomial time with access to a sparse* NP*-oracle is simulated by a propositional proof system with logarithmic advice.*

Proof. For the first claim, let f be a propositional proof system computed by the polynomial-time Turing transducer M_f with advice function h_f where $|h_f(n)| \leq c \cdot \log n$ for some constant c. Without loss of generality, we may assume that f uses input advice (cf. [2]). We choose a length-injective polynomial-time computable pairing function $\langle \cdot \rangle$ and consider the set

$$A = \left\{ \langle 1^n, a \rangle \mid a \in \Sigma^{\leq c \cdot \log n} \text{ and for some } \pi \in \Sigma^n,\ M_f(\pi, a) \notin \text{TAUT} \right\} \ ,$$

where $M_f(\pi, a)$ denotes the computation of M_f on input π under advice a. Intuitively, A collects all incorrect advice strings for M_f on length n. By construction, A is sparse. Further, $A \in \text{NP}$ because on input $\langle 1^n, a \rangle$ we can guess $\pi \in \Sigma^n$ and nondeterministically verify $M_f(\pi, a) \notin \text{TAUT}$ by guessing a satisfying assignment for $\neg M_f(\pi, a)$.

We now construct a polynomial-time oracle Turing transducer M_g which under oracle A computes a proof system $g \geq f$. Proofs in g will be of the form $\langle \pi, \varphi \rangle$. On such input, M_g queries all strings $\langle 1^{|\pi|}, a \rangle$, $a \in \Sigma^{\leq c \cdot \log |\pi|}$. For each negative answer, M_g simulates M_f on input π using a as advice. If any of these simulations outputs φ, then M_g also outputs φ, otherwise $g(\langle \pi, \varphi \rangle)$ is undefined.

Because M_g performs at most polynomially many simulations of M_f, the machine M_g runs in polynomial time. Correctness and completeness of g follow from the fact that M_f is simulated with all correct advice strings, and the original advice used by M_f is among these (as also other advice strings are used, g might have shorter proofs than f, though).

For the second claim, let f be a propositional proof system computed by the oracle transducer M_f under the sparse NP-oracle A. Let M_A be an NP-machine for A and let $p(n)$ be a polynomial bounding the cardinality of $A \cap \Sigma^{\leq n}$ as well as the running times of M_A and M_f. With these conventions, there are at most $q(n) = p(p(n))$ many strings in A that M_f may query on inputs of length n.

We now define a machine M_g, an advice function h_g, and an advice selector ℓ_g which together yield a propositional proof system $g \geq f$ with logarithmic advice. The advice function will be $h_g(n) = |A \cap \Sigma^{\leq p(n)}|$. As A is sparse this results in logarithmic advice. Proofs in the system g are of the form

$$\pi_g = \langle a_1, w_1, \ldots, a_{q(n)}, w_{q(n)}, \pi_f \rangle$$

where $\pi_f \in \Sigma^n$ (an f-proof), $a_1, \ldots, a_{q(n)} \in \Sigma^{\leq p(n)}$ (elements from A), and $w_1, \ldots, w_{q(n)} \in \Sigma^{\leq q(n)}$ (computations of M_A). At such a proof π_g, the advice selector chooses the advice corresponding to $|\pi_f|$, i.e., we set $\ell_g(\pi_g) = |\pi_f|$. The machine M_g works as follows: it first uses its advice to obtain the number $m = h_g(|\pi_f|)$ and checks whether $a_1, \ldots, a_m$ are pairwise distinct and for each $i = 1, \ldots, m$, the string w_i is an accepting computation of M_A on input a_i. If all these simulations succeed, then we know that $A \cap \Sigma^{\leq p(n)} = \{a_1, \ldots, a_m\}$. Hence M_g can now simulate M_f on π_f and give correct answers to all oracle queries made in this computation. □

Let us remark that Balcázar and Schöning [1] have shown a similar trade-off between advice and oracle access in complexity theory: $\mathsf{coNP} \subseteq \mathsf{NP/log}$ if and only if $\mathsf{coNP} \subseteq \mathsf{NP}^S$ for some sparse $S \in \mathsf{NP}$. We complete the picture by showing that the simulations in the previous theorem cannot be strengthened to a full equivalence between the two concepts:

Proposition 4. *There exist propositional proof systems with constant advice which cannot be computed with access to a recursive oracle.*

Proof. Let f be a polynomial-time computable propositional proof system. With each infinite sequence $a = (a_i)_{i \in \mathbb{N}}$, $a_i \in \{0, 1\}$, we associate the system

$$f_a(\pi) = \begin{cases} f(\pi') & \text{if either } \pi = 0\pi' \text{ or } (\pi = 1\pi' \text{ and } a_{|\pi|} = 0) \\ \text{undefined} & \text{if } \pi = 1\pi' \text{ and } a_{|\pi|} = 1. \end{cases}$$

As different sequences a and b yield different proof systems f_a and f_b, there exist uncountably many different propositional proof systems with one bit of advice. On the other hand, there are only countably many proof systems computed by oracle Turing machines under recursive oracles. Hence the claim follows. □

3 Optimal Proof Systems

A propositional proof system which simulates every other propositional proof system is called *optimal*. While in the classical setting, the existence of optimal proof systems is a prominent open question [8], Cook and Krajíček [5] have shown that there exists a propositional proof system with one bit of input advice which simulates all classical Cook-Reckhow proof systems. Combining this result with Theorem 3 yields:

Corollary 5. *There exists a propositional proof system f which simulates every polynomial-time computable propositional proof system. The system f is computable in polynomial time under a sparse* NP*-oracle.*

Our next result shows that if advice does not help to shorten proofs even for easy languages, then optimal propositional proof systems exist.

Theorem 6. *If every polynomially bounded proof system with logarithmic output advice for some $L \in$ coNP can be simulated by a proof system without advice, then the class of all polynomial-time computable propositional proof systems contains an optimal system.*

Proof. The classical Cook-Reckhow Theorem characterizes the existence of polynomially bounded proof systems: a language L has a polynomially bounded polytime computable proof system if and only if $L \in \mathsf{NP}$. This result also holds in the presence of advice (cf. [5,2]): L has a polynomially bounded proof system with logarithmic output advice if and only if $L \in \mathsf{NP/log}$. For languages from coNP, this equivalence even holds for input instead of output advice [2]. Therefore, we can restate the hypothesis of the theorem as $(\mathsf{coNP} \cap \mathsf{NP/log}) \subseteq \mathsf{NP}$.

Now we can apply a result of Balcázar and Schöning [1] who have shown that $(\mathsf{coNP} \cap \mathsf{NP/log}) \subseteq \mathsf{NP}$ holds if and only if $\mathsf{NE} = \mathsf{coNE}$. The latter condition, however, is known to imply the existence of optimal propositional proof systems in the classical sense, as shown by Krajíček and Pudlák [8] (cf. also [7]). □

Let us remark that the hypothesis in Theorem 6 does not refer to TAUT, but only to some of its subsets which are easy to prove with the help of advice.

4 Simplifying the Advice

There are two natural ways to enhance proof systems with advice by either supplying non-uniform information to the proof (input advice) or to the proven formula (output advice). Intuitively, input advice is the stronger model: proofs can be quite long and formulas of the same size typically require proofs of different size. Hence, supplying advice depending on the proof size is not only more flexible, but also results in more advice per formula. This view is also supported by previous results: there exist optimal proof systems with input advice [5,2], whereas for output advice a similar result cannot be obtained by current techniques [3]. Further evidence is provided by the existence of languages that have

polynomially bounded proof systems with logarithmic input advice, but do not have such systems with output advice [2].

In our next result we show how input advice can be transformed into output advice. We obtain this simplification of advice under the assumption of weak, but non-trivial upper bounds to the proof size. More precisely, from a proof system which uses logarithmic input advice and has sub-exponential size proofs of all tautologies, we construct a system with polynomial output advice which obeys almost the same upper bounds. For the proof we use a new technique by Buhrman and Hitchcock [4] who show that sets of sub-exponential density are not NP-hard unless coNP $\subseteq$ NP/poly.

Theorem 7. *Let $t(n) \in 2^{O(\sqrt{n})}$ and assume that there exists a $t(n)$-bounded propositional proof system f with polylogarithmic input advice. Then there exists an $s(n)$-bounded propositional proof system g with polynomial output advice where $s(n) \in O(t(d \cdot n^2))$ with some fixed constant d independent of f.*

Proof. Let $t(n) \leq 2^{c \cdot \sqrt{n}}$ for some constant c and let f be a $t(n)$-bounded propositional proof system with polylogarithmic input advice. We say that π is a *conjunctive f-proof* for a tautology φ if there exist tautologies $\psi_1, \ldots, \psi_n$ with $|\psi_i| = |\varphi| = n$ such that $f(\pi) = \psi_1 \wedge \cdots \wedge \psi_n$ and φ is among the ψ_i. For a number $m \geq 1$, we denote by $\sharp^n_m$ the number of tautologies $\varphi \in \mathrm{TAUT}^{=n}$ which have conjunctive f-proofs of size exactly m. By counting we obtain

$$(\sharp^n_m)^n \geq |\{(\varphi_1, \ldots, \varphi_n) \mid \varphi_1 \wedge \cdots \wedge \varphi_n \text{ has an } f\text{-proof of size } m \text{ and } |\varphi_i| = n \text{ for } 1 \leq i \leq n\}| \ . \tag{1}$$

As f is t-bounded, every $\varphi \in \mathrm{TAUT}^{=n}$ has a conjunctive f-proof of size at most $t(d \cdot n^2)$ where d is a constant such that for each sequence $\psi_1, \ldots, \psi_n$ of formulas of length n, $|\psi_1 \wedge \cdots \wedge \psi_n| \leq d \cdot n^2$. Let $\sharp^n = \max\{\sharp^n_m \mid m \leq t(d \cdot n^2)\}$. Using (1) we obtain

$$\begin{aligned} |\mathrm{TAUT}^{=n}|^n &\leq \sum_{m=1}^{t(d \cdot n^2)} (\sharp^n_m)^n \ \leq \ (\sharp^n)^n \cdot t(d \cdot n^2) \\ &\leq (\sharp^n)^n \cdot 2^{c \cdot \sqrt{d \cdot n^2}} \ = \ (\sharp^n \cdot 2^{c \cdot \sqrt{d}})^n \ . \end{aligned}$$

Thus, setting $\delta = 2^{-c \cdot \sqrt{d}}$, we get $\sharp^n \geq \delta \cdot |\mathrm{TAUT}^{=n}|$. Therefore, by definition of $\sharp^n$ there exists a number $m_{n,0} \leq t(d \cdot n^2)$ such that $\sharp^n_{m_{n,0}} \geq \delta \cdot |\mathrm{TAUT}^{=n}|$, i.e., a δ-fraction of all tautologies of length n has a conjunctive f-proof of size $m_{n,0}$.

Consider now the set $\mathrm{TAUT}^{=n}_0$ of all tautologies of length n which do not have conjunctive f-proofs of size $m_{n,0}$. Repeating the above argument for $\mathrm{TAUT}^{=n}_0$ yields a proof length $m_{n,1}$ such that $\sharp^n_{m_{n,1}} \geq \delta \cdot |\mathrm{TAUT}^{=n}_0|$. Iterating this argument we obtain a sequence

$$m_{n,0}, m_{n,1}, \ldots, m_{n,\ell(n)} \qquad \text{with } \ell(n) = \left\lceil \frac{\log |\mathrm{TAUT}^{=n}|}{\log(1-\delta)^{-1}} \right\rceil \leq \left\lceil \frac{n}{\log(1-\delta)^{-1}} \right\rceil$$

such that every $\varphi \in \mathrm{TAUT}^{=n}$ has a conjunctive f proof of size $m_{n,i}$ for some $i \in \{0, \ldots, \ell(n)\}$.

We will now define a proof system g which uses polynomial output advice and obeys the same proof lengths as f. Assume that f is computed by the polynomial-time Turing transducer M_f with advice function h_f. The system g will be computed by a polynomial-time Turing transducer M_g using the advice function

$$h_g(n) = \langle m_{n,0}, h_f(m_{n,0}), \ldots, m_{n,\ell(n)}, h_f(m_{n,\ell(n)}) \rangle \quad .$$

The machine M_g works as follows: first M_g checks whether the proof has the form

$$\langle \varphi, \psi_1, \ldots, \psi_n, \pi, i \rangle$$

where $\varphi, \psi_1, \ldots, \psi_n$ are formulas of length n such that $\varphi \in \{\psi_1, \ldots, \psi_n\}$, π is a string (an f-proof), and i is a number $\leq \ell(n)$. If this test fails, then M_g outputs $\top^n$ (an easy tautology of length n). Then M_g uses its advice to check whether $|\pi| = m_{n,i}$. If so, then M_g simulates M_f on input π using advice $h_f(m_{n,i})$ (which is available through the advice function h_g). If the output of this simulation is $\psi_1 \wedge \cdots \wedge \psi_n$, then M_g outputs φ, otherwise it outputs $\top^n$.

By our analysis above, g is a propositional proof system (it is correct and complete). The advice only depends on the length n of the proven formula, so g uses output advice. To estimate the advice length, let $|h_f(m)| \leq \log^a m$ for some constant a. Then

$$|h_g(n)| \leq \sum_{i=0}^{\ell(n)} (|m_{n,i}| + |h(m_{n,i}|) \leq (\ell(n)+1) \left(c\sqrt{n} + \log^a(2^{c\sqrt{n}}) \right) \in n^{O(1)} \quad .$$

The size of a g-proof $\langle \varphi, \psi_1, \ldots, \psi_n, \pi, i \rangle$ for $\varphi \in \mathrm{TAUT}^{=n}$ is dominated by $|\pi| \leq t(d \cdot n^2)$, and therefore g is $s(n)$-bounded for some $s(n) \in O(t(d \cdot n^2))$. □

5 Conclusion

Does advice help to prove propositional tautologies? In this generality, we leave open the question – but our results provide partial answers. On the one hand, when proving tautologies "very complicated" advice is not necessary – it suffices to use a "small amount of simple" advice (Theorem 3). Further, if advice is helpful to prove tautologies in the sense that proofs become shorter in general, then again the advice can be simplified (Theorem 7).

On the other hand, if advice is not at all useful to prove tautologies, then optimal propositional proof systems exist (Theorem 6), a consequence which is considered unlikely by many researchers (cf. [7]). For further research, it seems interesting to explore how natural proof systems like resolution can facilitate advice. Is it possible to shorten proofs in such systems by using advice?

Acknowledgement. The first author wishes to thank Uwe Schöning for suggesting to apply results from [1] in the context of proof systems with advice.

References

1. Balcázar, J., Schöning, U.: Logarithmic advice classes. Theoretical Computer Science 99, 279–290 (1992)
2. Beyersdorff, O., Köbler, J., Müller, S.: Nondeterministic instance complexity and proof systems with advice. In: Proc. 3rd International Conference on Language and Automata Theory and Applications. LNCS, vol. 5457, pp. 164–175. Springer, Heidelberg (2009)
3. Beyersdorff, O., Müller, S.: A tight Karp-Lipton collapse result in bounded arithmetic. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 199–214. Springer, Heidelberg (2008)
4. Buhrman, H., Hitchcock, J.M.: NP-hard sets are exponentially dense unless coNP $\subseteq$ NP/poly. In: Proc. 23rd Annual IEEE Conference on Computational Complexity, pp. 1–7 (2008)
5. Cook, S.A., Krajíček, J.: Consequences of the provability of NP $\subseteq$ P/poly. The Journal of Symbolic Logic 72(4), 1353–1371 (2007)
6. Cook, S.A., Reckhow, R.A.: The relative efficiency of propositional proof systems. The Journal of Symbolic Logic 44(1), 36–50 (1979)
7. Köbler, J., Messner, J., Torán, J.: Optimal proof systems imply complete sets for promise classes. Information and Computation 184(1), 71–92 (2003)
8. Krajíček, J., Pudlák, P.: Propositional proof systems, the consistency of first order theories and the complexity of computations. The Journal of Symbolic Logic 54(3), 1063–1079 (1989)

Backdoors in the Context of Learning

Bistra Dilkina, Carla P. Gomes, and Ashish Sabharwal

Department of Computer Science
Cornell University, Ithaca NY 14853-7501, U.S.A.
{bistra,gomes,sabhar}@cs.cornell.edu

Abstract. The concept of backdoor variables has been introduced as a structural property of combinatorial problems that provides insight into the surprising ability of modern satisfiability (SAT) solvers to tackle extremely large instances. This concept is, however, oblivious to "learning" during search—a key feature of successful combinatorial reasoning engines for SAT, mixed integer programming (MIP), etc. We extend the notion of backdoors to the context of learning during search. We prove that the smallest backdoors for SAT that take into account clause learning and order-sensitivity of branching can be exponentially smaller than "traditional" backdoors. We also study the effect of learning empirically.

1 Introduction

In recent years we have seen tremendous progress in the state of the art of SAT solvers: we can now efficiently solve large real-world problems. A fruitful line of research in understanding and explaining this outstanding success focuses on the role of *hidden structure* in combinatorial problems. One example of such hidden structure is a backdoor set, i.e., a set of variables such that once they are instantiated, the remaining problem *simplifies* to a tractable class [6–8, 12, 15, 16]. Backdoor sets are defined with respect to efficient sub-algorithms, called *sub-solvers*, employed within the systematic search framework of SAT solvers. In particular, the definition of strong backdoor set B captures the fact that a systematic tree search procedure (such as DPLL) restricted to branching only on variables in B will successfully solve the problem, whether satisfiable or unsatisfiable. Furthermore, in this case, the tree search procedure restricted to B will succeed independently of the order in which it explores the search tree.

Most state-of-the-art SAT solvers rely heavily on clause learning which adds new clauses every time a conflict is derived during search. Adding new information as the search progresses has not been considered in the traditional concept of backdoors. In this work we extend the concept of backdoors to the context of learning, where information learned from previous search branches is allowed to be used by the sub-solver underlying the backdoor. This often leads to much smaller backdoors than the "traditional" ones. In particular, we prove that the smallest backdoors for SAT that take into account clause learning can be exponentially smaller than traditional backdoors oblivious to these solver features. We also present empirical results showing that the added power of learning-sensitive backdoors is also often observed in practice.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 73–79, 2009.

2 Preliminaries

For lack of space, we will assume familiarity with Boolean formulas in conjunctive normal form (CNF), the satisfiability testing problem (SAT), and DPLL-based backtrack search methods for SAT. *Backdoor sets* for such formulas and solvers are defined with respect to efficient sub-algorithms, called *sub-solvers*, employed within the systematic search framework of SAT solvers. In practice, these sub-solvers often take the form of efficient procedures such as unit propagation (UP), pure literal elimination, and failed-literal probing. In some theoretical studies, solution methods for structural sub-classes of SAT such as 2-SAT, Horn-SAT, and RenamableHorn-SAT have also been considered as sub-solvers. Formally [16], a *sub-solver A for SAT* is any polynomial time algorithm satisfying certain natural properties on every input CNF formula F: (1) Trichotomy: A either determines F correctly (as satisfiable or unsatisfiable) or fails; (2) A determines F for sure if F has no clauses or contains the empty clause; and (3) if A determines F, then A also determines $F|_{x=0}$ and $F|_{x=1}$ for any variable x.

For a formula F and a truth assignment τ to a subset of the variables of F, we will use $F|_\tau$ to denote the simplified formula obtained after applying the (partial) truth assignment to the affected variables.

Definition 1 (Weak and Strong Backdoors for SAT [16]). *Given a CNF formula F on variables X, a subset of variables $B \subseteq X$ is a* weak backdoor *for F w.r.t. a sub-solver A if for* some *truth assignment $\tau : B \to \{0,1\}$, A returns a satisfying assignment for $F|_\tau$. Such a subset B is a* strong backdoor *if for* every *truth assignment $\tau : B \to \{0,1\}$, A returns a satisfying assignment for $F|_\tau$ or concludes that $F|_\tau$ is unsatisfiable.*

Weak backdoor sets capture the fact that a well-designed heuristic can get "lucky" and find the solution to a hard satisfiable instance if the heuristic guidance is correct even on the small fraction of variables that constitute the backdoor set. Similarly, strong backdoor sets B capture the fact that a systematic tree search procedure (such as DPLL) restricted to branching only on variables in B will successfully solve the problem, whether satisfiable or unsatisfiable. Furthermore, in this case, the tree search procedure restricted to B will succeed independently of the order in which it explores the search tree.

3 Backdoor Sets for Clause Learning SAT Solvers

The last point made in Section 2—that the systematic search procedure will succeed independent of the order in which it explores various truth valuations of variables in a backdoor set B—is, in fact, a very important notion that has only recently begun to be investigated, in the context of mixed-integer programming [1]. In practice, many modern SAT solvers employ *clause learning* techniques, which allow them to carry over information from previously explored branches to newly considered branches. Prior work on proof methods based on clause learning and the resolution proof system suggests that, especially for

unsatisfiable formulas, some variable-value assignment orders may lead to significantly shorter search proofs than others. In other words, it is very possible that "learning-sensitive" backdoors are much smaller than "traditional" strong backdoors. To make this notion of incorporating learning-during-search into backdoor sets more precise, we introduce the following extended definition:

Definition 2 (Learning-Sensitive Backdoors for SAT). *Given a CNF formula F on variables X, a subset of variables $B \subseteq X$ is a* learning-sensitive backdoor *for F w.r.t. a sub-solver A if there exists a search tree exploration order such that a clause learning SAT solver branching only on the variables in B, with this order and with A as the sub-solver at the leaves of the search tree, either finds a satisfying assignment for F or proves that F is unsatisfiable.*

Note that, as before, each leaf of this search tree corresponds to a truth assignment $\tau : B \to \{0, 1\}$ and induces a simplified formula $F|_\tau$ to be solved by A. However, the tree search is naturally allowed to carry over and use learned information from previous branches in order to help A determine $F|_\tau$. Thus, while $F|_\tau$ may not always be solvable by A *per se*, additional information gathered from previously explored branches may help A solve $F|_\tau$. We note that incorporating learned information can, in principle, also be considered for the related notion of *backdoor trees* [14], which looks at the smallest search tree size rather than the set of branching variables.

We explain the power of learning-sensitivity through the following example formula, for which there is a natural learning-sensitive backdoor of size one w.r.t. unit propagation but the smallest traditional strong backdoor is of size 2. We will then generalize this observation into an exponential separation between the power of learning-sensitive and traditional strong backdoors for SAT.

Example 1. Consider the unsatisfiable SAT instance, F_1:

$$(x \vee p_1), (x \vee p_2), (\neg p_1 \vee \neg p_2 \vee q), \quad (\neg q \vee a), (\neg q \vee \neg a \vee b), (\neg q \vee \neg a \vee \neg b)$$
$$(\neg x \vee q \vee r), \quad (\neg r \vee a), (\neg r \vee \neg a \vee b), (\neg r \vee \neg a \vee \neg b)$$

We claim that $\{x\}$ is a learning-sensitive backdoor for F_1 w.r.t. the unit propagation sub-solver, while all traditional strong backdoors are of size at least two. First, let's understand why $\{x\}$ does work as a backdoor set when clause learning is allowed. When we set $x = 0$, this implies—by unit propagation—the literals p_1 and p_2, these together imply q which implies a, and finally, q and a together imply both b and $\neg b$, causing a contradiction. At this point, a clause learning algorithm will realize that the literal q forms what's called a unique implication point (UIP) for this conflict [10], and will learn the singleton clause $\neg q$. Now, when we set $x = 1$, this, along with the learned clause $\neg q$, will unit propagate one of the clauses of F_1 and imply r, which will then imply a and cause a contradiction as before. Thus, setting $x = 0$ leads to a contradiction by unit propagation as well as a learned clause, and setting $x = 1$ after this also leads to a contradiction.

To see that there is no traditional strong backdoor of size one with respect to unit propagation (and, in particular, $\{x\}$ does not work as a strong backdoor

without the help of the learned clause $\neg q$), observe that for every variable of F_1, there exists at least one polarity in which it does not appear in any 1- or 2-clause (i.e., a clause containing only 1 or 2 variables) and therefore there is no empty clause generation or unit propagation under at least one truth assignment for that variable. (Note that F_1 does not have any 1-clauses to begin with.) E.g., q does not appear in any 2-clause of F_1 and therefore setting $q = 0$ does not cause any unit propagation at all, eliminating any chance of deducing a conflict. Similarly, setting $x = 1$ does not cause any unit propagation. In general, no variable of F_1 can lead to a contradiction by itself under both truth assignments to it, and thus cannot be a traditional strong backdoor. Note that $\{x, q\}$ does form a traditional strong backdoor of size two for F_1 w.r.t. unit propagation. □

Theorem 1. *There are* unsatisfiable *SAT instances for which the smallest learning-sensitive backdoors w.r.t. unit propagation are exponentially smaller than the smallest traditional strong backdoors.*

Proof (Sketch). We, in fact, provide two proofs of this statement by constructing two unsatisfiable formulas F_2 and F_3 over $N = k+3\cdot 2^k$ variables and $M = 4\cdot 2^k$ clauses, with the following property: both formulas have a learning-sensitive backdoor of size $k = \Theta(\log N)$ but no traditional strong backdoor of size smaller than $2^k + k = \Theta(N)$. F_2 is perhaps a bit easier to understand and has a relatively weak ordering requirement for the size k learning-sensitive backdoor to work (namely, that the all-1 truth assignment must be evaluated at the very end); F_3, on the other hand, requires a strict value ordering to work as a backdoor (namely, the lexicographic order from $000\ldots0$ to $111\ldots1$) and highlights the strong role a good branching order plays in the effectiveness of backdoors. For lack of space, the details are deferred to an extended Technical Report [3]. □

In fact, the discussion in the proof of Theorem 1 also reveals that for the constructed formula F_3, any value ordering that starts by assigning 1's to all x_i's will lead to a learning-sensitive backdoor of size no smaller than 2^k. This immediately yields the following result under-scoring the importance of the "right" value ordering even amongst various learning-sensitive backdoors.

Corollary 1. *There are* unsatisfiable *SAT instances for which one value ordering of the variables can lead to exponentially smaller learning-sensitive backdoors w.r.t. unit propagation than a different value ordering.*

We now turn our attention to the study of strong backdoors for *satisfiable* instances, and show that clause learning can also lead to strictly smaller (strong) backdoors for satisfiable instances. In fact, our experiments suggest a much more drastic impact of clause learning on backdoors for practical satisfiable instances than on backdoors for unsatisfiable instances. We have the following formal result that can be derived from a slight modification of the construction of formula F_1 used earlier in Example 1 (see Technical Report [3]).

Theorem 2. *There are* satisfiable *SAT instances for which there exist learning-sensitive backdoors w.r.t. unit propagation that are smaller than the smallest traditional strong backdoors.*

As a closing remark, we note that the presence of clause learning does not affect the power of weak backdoors w.r.t. a natural class of *syntactically-defined* sub-solvers, i.e., sub-solvers that work when the constraint graph of the instance satisfies a certain polynomial-time verifiable property. Good examples of such syntactic classes w.r.t. which strong backdoors have been studied in depth are 2-SAT, Horn-SAT, and RenamableHorn-SAT [cf. 2, 11, 12]. Most of such syntactic classes satisfy a natural property, namely, they are *closed under clause removal*. In other words, if F is a 2-SAT or Horn formula, then removing some clauses from F yields a smaller formula that is also a 2-SAT or Horn formula, respectively. We have the following observation (see Technical Report [3] for a proof):

Proposition 1. *Clause learning does not reduce the size of weak backdoors with respect to syntactic sub-solver classes that are closed under clause removal.*

4 Experimental Results

We evaluate the effect of clause learning on the size of backdoors in a set of well-known SAT instances from SATLIB [5]. Upper bounds on the size of the smallest leaning-sensitive backdoor w.r.t. UP were obtained using the SAT solver *RSat* [13]. At every search node *RSat* employs UP and at every conflict it employs clause learning based on UIP. We turned off restarts and randomized the variable and value selection. In addition, we traced the set of variables used for branching during search—the backdoor. We ran the modified *RSat* 5,000 times per instance and recorded the smallest backdoor set among all runs.

Upper bounds on the size of the smallest traditional backdoor w.r.t. UP were obtained using a modified version of *Satz-rand* [4, 9] that employs UP as a sub-solver and also traces the set of branch variables. We ran the modified *Satz* 5,000 times per instance and recorded the smallest backdoor set among all runs. Note that these results concern traditional weak backdoors for satisfiable instances and strong backdoors for unsatisfiable instances. *Satz* relies heavily on good variable selection heuristics in order to minimize the solution time. Hence, using

Table 1. Upper bounds on the size of the smallest backdoor when using clause learning and unit propagation (within RSat) and when using only unit propagation (within Satz). Results are given as percentage of the number of variables.

Instance	Status	Vars	Clauses	**UP+CL**	UP
bf0432-007	UNSAT	1,040	3,668	**12.12%**	13.65%
bf1355-075	UNSAT	2,180	6,778	**3.90%**	5.92%
bf1355-638	UNSAT	2,177	6,768	**3.86%**	6.84%
bf2670-001	UNSAT	1,393	3,434	**1.22%**	2.08%
apex7_gr_2pin_w4	UNSAT	1,322	10,940	**12.25%**	20.73%
parity_unsat_4_5	UNSAT	2,508	17,295	**9.85%**	39.07%
anomaly	SAT	48	261	4.17%	4.17%
medium	SAT	116	953	**1.72%**	14.66%
huge	SAT	459	7,054	**1.09%**	3.27%
bw_large.a	SAT	459	4,675	**1.53%**	3.49%
bw_large.b	SAT	1,087	13,772	**1.93%**	11.59%
bw_large.c	SAT	3,016	50,457	**2.95%**	13.76%
bw_large.d	SAT	6,325	131,973	**3.37%**	43.27%

Satz instead of a modified version of *RSat* with learning turned off gave us much better bounds on traditional backdoors w.r.t. UP.

The results are summarized in Table 1. Across all satisfiable instances the learning-sensitive backdoor upper bounds are significantly smaller than the traditional ones. For unsatisfiable instances, the upper bounds on the learning-sensitive and traditional backdoors are not very different. However, a notable exception is the *parity* instance where including clause learning reduces the backdoor upper bound to less than 10% from almost 39%.

Acknowledgments

This research was supported by IISI, Cornell University (AFOSR grant FA9550-04-1-0151), NSF Expeditions in Computing award for Computational Sustainability (Grant 0832782) and NSF IIS award (Grant 0514429). The first author was partially supported by an NSERC PGS Scholarship. Part of this work was done while the third author was visiting McGill University.

References

[1] Dilkina, B., Gomes, C.P., Malitsky, Y., Sabharwal, A., Sellmann, M.: Backdoors to combinatorial optimization: Feasibility and optimality. In: CPAIOR (2009)
[2] Dilkina, B., Gomes, C.P., Sabharwal, A.: Tradeoffs in the complexity of backdoor detection. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 256–270. Springer, Heidelberg (2007)
[3] Dilkina, B., Gomes, C.P., Sabharwal, A.: Backdoors in the context of learning (extended version). Technical report, Cornell University, Computing and Information Science (April 2009), http://hdl.handle.net/1813/12231
[4] Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: AAAI, pp. 431–437 (1998)
[5] Hoos, H.H., Stützle, T.: SATLIB: An online resource for research on SAT. In: SAT, pp. 283–292 (2000), http://www.satlib.org
[6] Kilby, P., Slaney, J.K., Thiébaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: AAAI, pp. 1368–1373 (2005)
[7] Kullmann, O.: Investigating a general hierarchy of polynomially decidable classes of cnf's based on short tree-like resolution proofs. In: ECCC, vol. 41 (1999)
[8] Kullmann, O.: Upper and lower bounds on the complexity of generalised resolution and generalised constraint satisfaction problems. Annals of Mathematics and Artificial Intelligence 40(3-4), 303–352 (2004)
[9] Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: IJCAI, pp. 366–371 (1997)
[10] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC, pp. 530–535 (2001)
[11] Nishimura, N., Ragde, P., Szeider, S.: Detecting backdoor sets with respect to Horn and binary clauses. In: SAT, pp. 96–103 (2004)
[12] Paris, L., Ostrowski, R., Siegel, P., Sais, L.: Computing Horn strong backdoor sets thanks to local search. In: ICTAI, pp. 139–143 (2006)

[13] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: SAT, pp. 294–299 (2007)
[14] Samer, M., Szeider, S.: Backdoor trees. In: AAAI, pp. 363–368 (2008)
[15] Szeider, S.: Backdoor sets for DLL subsolvers. J. Auto. Reas. 35(1-3), 73–88 (2005)
[16] Williams, R., Gomes, C., Selman, B.: Backdoors to typical case complexity. In: IJCAI, pp. 1173–1178 (2003)

Solving SAT for CNF Formulas with a One-Sided Restriction on Variable Occurrences

Daniel Johannsen[1], Igor Razgon[2], and Magnus Wahlström[1]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] Cork Constraint Computation Centre, University College Cork, Ireland

Abstract. In this paper we consider the class of boolean formulas in Conjunctive Normal Form (CNF) where for each variable all but at most d occurrences are either positive or negative. This class is a generalization of the class of CNF formulas with at most d occurrences (positive and negative) of each variable which was studied in [Wahlström, 2005].

Applying complement search [Purdom, 1984], we show that for every d there exists a constant $\gamma_d < 2 - \frac{1}{2d+1}$ such that satisfiability of a CNF formula on n variables can be checked in runtime $O(\gamma_d^n)$ if all but at most d occurrences of each variable are either positive or negative. We thoroughly analyze the proposed branching strategy and determine the asymptotic growth constant γ_d more precisely. Finally, we show that the trivial $O(2^n)$ barrier of satisfiability checking can be broken even for a more general class of formulas, namely formulas where the positive or negative literals of every variable have what we will call a *d–covering*.

To the best of our knowledge, for the considered classes of formulas there are no previous non-trivial upper bounds on the complexity of satisfiability checking.

1 Introduction

Design of fast exponential algorithms for satisfiability checking has attracted considerable attention of various research communities in applied as well as in theoretical fields of computer science. Since it is unknown how to break the trivial $O(2^n)$ barrier for the runtime of the unrestricted satisfiability problem (SAT) on n variables, current research concentrates on more efficient satisfiability checking of restricted classes of conjunctive normal form (CNF) formulas. The most widely considered restriction is k–SAT, including all formulas whose maximal clause length is at most k. Currently the best runtime is $O\big((2-\frac{2}{k+1})^n\big)$ for a deterministic algorithm [3], and a stronger but comparable bound for a randomized one [4]. However, to improve the understanding of how the structure of a CNF formula influences the efficiency of satisfiability checking, it is important to study further sub-classes of CNF formulas. Two such sub-classes are formulas with restrictions on (i) the value of the density m/n (where m is the number of clauses) and (ii) on the number of occurrences of each variable. Calabro, Impagliazzo, and Paturi [2] proved that for both classes satisfiability can be checked in runtime $O(\gamma^n)$ with $\gamma < 2$, and asymptotically related the bounds of k–SAT, SAT with $m/n \leq \Delta$, and SAT with at most d occurrences per variable to one another, essentially showing that the two latter bounds behave

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 80–85, 2009.

as the former one with $k = \Theta(\log \Delta)$ and $k = \Theta(\log d)$, respectively. For small values of d, Wahlström [7,6] has given a stronger bound for formulas with at most d occurrences per variable of order $O(1.1279^{(d-2)n})$.

The main contribution of this paper is extending the boundaries of the classes of CNF formulas for which satisfiability can be checked faster than in runtime $O(2^n)$. In particular, we continue the line of research of [7], where the total number of occurrences (positive *and* negative) of each variable in a CNF formula F was restricted to at most d. Now, for each variable x of F we call the literal $\ell \in \{x, \neg x\}$ with less occurrences in F *minor* and its negation *major*. We then only restrict the number of minor literals in F to be at most d, that is, the total number of literal occurrences per variable in F remains unrestricted.

We first study the satisfiability of formulas where for each variable all but one occurrences are positive or negative and give an algorithm with runtime $O^*(3^{n/3})$ (Theorem 1)[1]. Then, we propose an algorithm based on complement search [5] and show that for each fixed d there exists a constant $\gamma_d < 2$ such that the satisfiability of CNF formulas with at most d minor literals per variable can be checked in runtime $O(\gamma_d^n)$ (Theorem 2). Next, we investigate the main parameter of the algorithm closer and bound γ_d by $2 - \frac{1}{2d+1}$ (Theorem 3). Finally, we further generalize the class of CNF formulas with at most d minor literals per variable to a class we call *CNF formulas with a d–covering* and present an algorithm checking satisfiability of such formulas in runtime $O^*(\Phi_{d+1}^n)$ (Theorem 4), where $\Phi_d^n < 2$ is the Fibonacci constant of order d as defined below.

2 SAT for CNF Formulas with Unique Minor Literals

In this section we study CNF formulas with exactly one minor literal per variable.

Lemma 1. *A CNF formula with at most one minor literal per variable either contains a clause consisting only of minor literals or is trivially satisfied by satisfying all major literals.*

This basic observation allows us to check the satisfiability of such formulas.

Algorithm 1
Input: *CNF formula F with set of minor literals M, each occurring once.*
Output: TRUE *if F is satisfiable, otherwise* FALSE.
If *F is empty* **then** *return* TRUE.
If *F contains an empty clause* **then** *return* FALSE.
If *F contains a clause $C = (\ell_1 \vee \ldots \vee \ell_r)$ with $\ell_1, \ldots, \ell_r \in M$* **then**
  let *$C_i = C[\ell_i = \text{FALSE}]$*[2] *for $i \in \{1, \ldots, r\}$* **and**
  try branches *$F[\ell_i = \text{TRUE}, C_i = \text{FALSE}]$ for $i \in \{1, \ldots, r\}$*
else *evaluate $F[\ell = \text{FALSE}$ for all $\ell \in M]$.*

[1] We use the $O^*(\cdot)$ notation to suppress factors polynomial in the length of the input.
[2] For a literal ℓ or a clause C we denote by $F[\ell\,/\,C = \text{TRUE}\,/\,\text{FALSE}]$ the residual formula obtained by assigning TRUE or FALSE to ℓ or all literals of C, respectively. If this assigns all literals of a clause to FALSE, then leave the clause as an empty clause (indicating that a contradiction has been encountered).

According to Lemma 1 this algorithm is correct. The runtime is dominated by the branching strategy and thus of order $\max_{r\in\mathbb{N}} r^{n/r}$ which is worst at $r = 3$.

Theorem 1. *Let $n \in \mathbb{N}$ and let F be a CNF formula on n variables with at most one minor literal per variable. Then Algorithm 1 checks the satisfiability of F in runtime $O^*(3^{n/3}) \subseteq O(1.4423^n)$.*

Correspondingly, we can express a pseudo-lower runtime bound of $\Omega(2^{n/2})$ which holds if the general SAT problem cannot be solved faster than in $\Omega(2^n)$.

Lemma 2. *Let $n \in \mathbb{N}$ and $\gamma > 1$. If the satisfiability of any CNF formula on n variables with at most one minor literal per variable can be checked in runtime $O(\gamma^n)$ then the satisfiability of any (unrestricted) CNF formula on n variables can be checked in runtime $O(\gamma^{2n})$.*

Proof. Every general CNF formula on n variables can be equivalently transformed to a CNF formula on $2n$ variables such that each minor literal occurs at most once. For each variable, replace all minor literals by a new variable and add a clause containing the minor literal and the negation of the new variable.

A similar reduction from 3-CSP improves this pseudo-lower bound to $\Omega(3^{n/3})$ but is omitted due to lack of space.

3 SAT of CNF Formulas with at Most *d* Minor Literals

The following lemma by Purdom [5] allows us to recursively check the satisfiability of CNF formulas with n variables and at most d minor literals per variable in runtime $O(\gamma^n)$ with $\gamma < 2$.

Lemma 3. *Let ℓ be a literal of a CNF formula F. Then either $F[\ell = \text{False}]$ is satisfiable or for all satisfying assignments of F there is a clause C containing ℓ such that all literals of C are assigned* False *except for ℓ which is assigned* True.

The previous lemma allows us to check the satisfiability of CNF formulas with at most d minor literals per variable by using a branching strategy parameterized by the *branching threshold* k for short clauses.

Algorithm 2
Parameter: *Branching threshold k for short clauses.*
Input: *CNF formula F with set of minor literals M*
Output: True *if F is satisfiable, otherwise* False.
If *F is empty* **then** *return* True.
If *F contains an empty clause* **then** *return* False.
If *F contains a clause $C = (\ell_1 \vee \ldots \vee \ell_r)$ with $r \leq k$* **then**
  try branches *$F[\ell_1 = \ldots = \ell_{i-1} = \text{False}, \ell_i = \text{True}]$ for $i \in \{1, \ldots, r\}$*
else
  pick *$\ell \in M$ contained in the clauses $C_1, \ldots, C_s$* **and**
  let *$C'_i = C_i[\ell = \text{False}]$ for $i \in \{1, \ldots, s\}$* **and**
  try branches *$F[\ell = \text{True}]$ and $F[\ell = \text{True}, C'_i = \text{False}]$ for $i \in \{1, \ldots, s\}$.*

Let $T(n)$ be the runtime of the algorithms branching procedure (where n is the number of variables). The runtime of the branch where the shortest clause is of size $r \leq k$ is of order $T(n-1) + \ldots + T(n-r)$ which is at most

$$T_A(n) = \sum_{i=1}^{k} T(n-i) \tag{1}$$

The growth constant of this recursion is known to be the k–th order Fibonacci number Φ_k (see, e. g., [8]). That is, $T_A(n) \in O^*(\Phi_k^n)$, where Φ_k is the unique solution of the equation $x^k(2-x) = 1$ in the interval $(1,2)$. The number Φ_2 is the golden ratio $(1+\sqrt{5})/2$, more initial values of Φ_k are given in Table 1.

The runtime of the branch where the shortest clause is of size at least $k+1$ is at most $T_B(n) = T(n-1) + s \cdot T(n-(k+1))$ where s is the number of clauses containing the eliminated literal. Thus, for $s \leq d$, this runtime is at most

$$T_B(n) = T(n-1) + d \cdot T(n-1-k)\,. \tag{2}$$

Hence, the maximum of $T_A(n)$ and $T_B(n)$ is an upper bound on the runtime $T(n)$ of Algorithm 2 with parameter k on CNF formulas with at most d minor literals per variable. Obviously, $T(n)$ is strongly influenced by the choice of k. For example, if we choose $k = d+1$, then $T_A(n)$ dominates $T_B(n)$.

Theorem 2. *Let $n, d \in \mathbb{N}$ and let F be a CNF formula on n variables with at most d minor literals per variable. Then Algorithm 2 with parameter $d+1$ checks the satisfiability of F in runtime $O^*(\Phi_{d+1}^n)$.*

In the remainder of this section we see, how to choose the parameter k optimally for every fixed $d \in \mathbb{N}$. Suppose that k is also fixed. Then $T(n)$ is of order γ^n, where $\gamma \in (1,2)$ is the smallest constant that satisfies both recursions (1) and (2).

Lemma 4. *Let $n, d, k \in \mathbb{N}$ with $d \geq 2$ and let F be a CNF formula on n variables with at most d minor literals per variable. Then Algorithm 2 with parameter k checks the satisfiability of F in runtime $O^*(\gamma^n)$ for all $\gamma \in (1,2)$ with*

$$\frac{d}{\gamma - 1} \leq \gamma^k \leq \frac{1}{2-\gamma}\,. \tag{3}$$

Proof. The statement follows by induction on n. □

A direct consequence of the previous lemma is that the minimal γ satisfying condition (3) dominates the growth constant of $T(n)$ for given d and k. Clearly, the condition is satisfied for every $d \geq 2$ and $k \in \mathbb{N}$ as γ tends to two. On the other hand, as γ tends to one, eventually one of the two inequalities is violated. Thus, a minimal γ satisfies at least one of the two equations with equality.

On the other hand, if there exists a k such that the corresponding γ satisfies both inequalities, then γ is optimal for all values of k (decreasing k violates the first inequality while increasing k violates the second one; in both cases we need to increase γ to satisfy condition (3) again). This situation occurs if $\gamma = 2 - \frac{1}{d+1}$. In this case the lower and the upper bound on γ^k both have the value $d+1$. For smaller values of γ, condition (3) can never be satisfied.

Lemma 5. *Let $d \in \mathbb{N}$ with $d \geq 2$. Then $2 - \frac{1}{d+1}$ is a lower bound on all γ satisfying condition (3) for any $k \in \mathbb{N}$.*

Note that this lower bound is not necessarily attained since k is integral. If we drop this condition, then for $k_d^* = \log(d+1)/(\log(2d+1) - \log(d+1))$ both inequalities in condition (3) become equalities. For $k \geq \lceil k_d^* \rceil$, the right inequality in condition (3) is violated for $\gamma = 2 - \frac{1}{d+1}$, while the left right inequality is satisfied. For $k \leq \lfloor k_d^* \rfloor$ the opposite holds. The further k is apart from k_d^*, the more we have to increase γ to satisfy the violated inequality. Thus one of $\lfloor k_d^* \rfloor$ and $\lceil k_d^* \rceil$ is optimal, depending for which the corresponding γ is smaller. Table 1 lists k_d, γ_A and γ_B for the initial values of d.

Lemma 6. *Let $d \geq 2$ and let $k_d^* = \frac{\log(d+1)}{\log(2d+1) - \log(d+1)}$. Moreover, let γ_A and γ_B be the unique solutions of $(2-\gamma_A)\gamma_A^{\lceil k_d^* \rceil} = 1$ and $(\gamma_B - 1)\gamma_B^{\lfloor k_d^* \rfloor} = d$ in the interval $(1,2)$. Then $\gamma_d = \min\{\gamma_A, \gamma_B\}$ satisfies condition (3) for $k_d \in \{\lceil k_d^* \rceil, \lfloor k_d^* \rfloor\}$ chosen respectively. Furthermore, in this γ_d is minimal for all k.*

Finally, we show a closed form upper bound for the runtime of Algorithm 2.

Theorem 3. *Let $n, d \in \mathbb{N}$ with $d \geq 2$ and k_d and γ_d as in Lemma 6. Then Algorithm 2 with parameter k_d checks the satisfiability of a CNF formula on n variables with at most d minor literals per variable in runtime $O\big((2 - \frac{1}{2d+1})^n\big)$.*

Proof. For $d \geq 2$, γ is smaller than $d/(\gamma - 1)$ and strictly smaller than $1/(2-\gamma)$ divided by $d/(\gamma - 1)$. Hence, there exists a $k \in \mathbb{N}$ such that $\gamma = 2 - \frac{1}{2d+1}$ satisfies condition (3). The statement follows from Lemma 4 and Lemma 5.

4 Further Generalization

We can further generalize Lemma 3 to obtain an algorithm of runtime $O(\gamma^n)$ with $\gamma < 2$ for a class of CNF formulas which neither have short clauses nor a one-sided restriction on the variable occurrences.

Table 1. Runtime bounds for Algorithm 2 with parameter k on CNF formulas with at most d minor literals per variable. For $d = 2, \ldots, 6$ the table shows the following values: the optimal choice of the branching threshold k_d and the corresponding relaxed real-valued optimum k_d^*; the two choices of the optimal growth constant γ_A and γ_B (Lemma 6) with the better one in bold face; the lower bound $2-1/(d+1)$ (Lemma 5), the upper bound $2-1/(2d+1)$ (Theorem 3), and the weak upper bound Φ_{d+1} (Theorem 2).

d	k_d	k_d^*	γ_A	γ_B	$2 - \frac{1}{d+1}$	$2 - \frac{1}{2d+1}$	Φ_{d+1}
2	2	2.15064	**1.69562**	1.83929	1.66667	1.80000	1.83929
3	3	2.47720	1.86371	**1.83929**	1.75000	1.85714	1.92756
4	3	2.73817	2.00000	**1.83929**	1.80000	1.88889	1.96595
5	3	2.95602	2.11634	**1.83929**	1.83333	1.90909	1.98358
6	3	3.14343	**1.88947**	1.92756	1.85714	1.92308	1.99196

A *covering* of a literal ℓ in a CNF formula F is a set L of literals, such that (i) no literal and its negation are both in L, (ii) ℓ is not in L, and (iii) all clauses of F containing ℓ also contain a literal of L. We say that F has a *d–covering* if one of the two literals corresponding to each variable has a covering of size d.

It is not hard to see that a CNF formula with clauses of size at least $d+1$ and at most d minor literals per variable has a d–covering. But, this weaker condition is still sufficient for breaking the $O(2^n)$ runtime barrier.[3]

Algorithm 3
Input: *CNF formula F with set of minor literals M*
Output: TRUE *if F is satisfiable, otherwise* FALSE.
If *F is empty* **then** *return* TRUE.
If *F contains an empty clause* **then** *return* FALSE.
Pick *literal ℓ_0 covered by $\ell_1, \ldots \ell_d$* **and**
try branches $F[\ell_0 = \ldots = \ell_{i-1} = \text{TRUE}, \ell_i = \text{FALSE}]$ *for* $i \in \{0, \ldots, d\}$

According to Lemma 3 and the definition of a d–covering of a CNF formula F, any satisfying assignment of F also satisfies the clause $(\neg\ell_0 \vee \neg\ell_1 \vee \cdots \vee \neg\ell_d)$.

Theorem 4. *Let $n, d \in \mathbb{N}$. Then Algorithm 3 checks the satisfiability of a CNF formula on n variables that has a d–covering in runtime $O^*(\Phi^n_{d+1})$.*

References

1. Alon, N., Moshkovitz, D., Safra, S.: Algorithmic construction of sets for k-restrictions. ACM Trans. Algorithms 2(2), 153–177 (2006)
2. Calabro, C., Impagliazzo, R., Paturi, R.: A duality between clause width and clause density for SAT. In: Annual IEEE Conference on Computational Complexity, pp. 252–260 (2006)
3. Dantsin, E., Goerdt, A., Hirsch, E.A., Kannan, R., Kleinberg, J.M., Papadimitriou, C.H., Raghavan, P., Schöning, U.: A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search. Theoretical Computer Science 289(1), 69–83 (2002)
4. Paturi, R., Pudlák, P., Saks, M.E., Zane, F.: An improved exponential-time algorithm for k-sat. J. ACM 52(3), 337–364 (2005)
5. Purdom, P.W.: Solving satisfiability with less searching. IEEE Trans. Pattern Anal. Machine Intell. 6(4), 510–513 (1984)
6. Wahlström, M.: An algorithm for the SAT problem for formulae of linear length. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 107–118. Springer, Heidelberg (2005)
7. Wahlström, M.: Faster exact solving of SAT formulae with a low number of occurrences per variable. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 309–323. Springer, Heidelberg (2005)
8. Wolfram, D.: Solving generalized Fibonacci recurrences. The Fibonacci Quarterly 36(2), 129–145 (1998)

[3] For fixed $d \in \mathbb{N}$ we can test for a d–covering of a CNF formula (and also find it) in runtime $O(n^{f(d)})$. However, Parameterized Complexity Theory suggests also a lower bound of $\Omega(n^d)$. Furthermore, the problem to compute a minimal covering of a given CNF formula is a generalization of the hitting set problem, which in polynomial time cannot be approximated better than within a factor $\Theta(\log n)$ unless P=NP [1].

On Some Aspects of Mixed Horn Formulas

Stefan Porschen*, Tatjana Schmidt, and Ewald Speckenmeyer

Institut für Informatik, Universität zu Köln,
Pohligstr. 1, D-50969 Köln, Germany
{porschen,schmidt,esp}@informatik.uni-koeln.de

Abstract. We consider various aspects of the Mixed Horn formula class (MHF). A formula $F \in$ MHF consists of a 2-CNF part P and a Horn part H. We propose that MHF has a central relevance in CNF, because many prominent NP-complete problems, e.g. Feedback Vertex Set, Vertex Cover, Dominating Set and Hitting Set can easily be encoded as MHF. Furthermore we show that SAT for some interesting subclasses of MHF remains NP-complete. Finally we provide algorithms for two of these subclasses solving SAT in a better running time than $O(2^{0.5284n}) = O((\sqrt[3]{3})^n)$ which is the best bound for MHF so far, over n variables. One of these subclasses consists of formulas, where the Horn part is negative monotone and the variable graph corresponding to the positive 2-CNF part P consists of disjoint triangles only. For this class we provide an algorithm and present the running times for the k-uniform cases, where $k \in \{3, 4, 5, 6\}$. Regarding the other subclass consisting of certain k-uniform linear mixed Horn formulas, we provide an algorithm solving SAT in time $O((\sqrt[k]{k})^n)$, for $k \geq 4$.

Keywords: Mixed Horn formula, satisfiability, polynomial time reduction, exact algorithm, NP-completeness.

1 Introduction

In recent time the interest in designing exact algorithms providing better upper time bounds than the trivial ones for NP-complete problems and their NP-hard optimization counterparts has increased. Of particular interest in this context is the investigation of exact algorithms for testing the satisfiability (SAT) of propositional formulas in conjunctive normal form (CNF). This interest stems from the fact that SAT is well known to be a fundamental NP-complete problem appearing naturally or via reduction as the abstract core of many application-relevant problems. In this context it turns out that reducing many classical NP-complete problems to SAT, formulas of a restricted structure are generated. Namely formulas $F = P \wedge H$ of a positive monotone 2-CNF part P and a Horn part H, called mixed Horn formulas (MHF) according to [8]. Therefore it is worthwhile to design good algorithms for solving formulas of this special structure.

* The first author was partially supported by the DFG project under grant No. SP 317/7-1.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 86–100, 2009.

As already shown in [8] graph colorability, problems for level graphs, like level-planarity test or the NP-hard crossing-minimization problem [10], can be formulated conveniently in terms of MHF. The main purpose of this paper is to provide a more systematical insight into MHF as destination class and thus to illustrate that MHF has a central relevance in CNF. To that end we provide straightforward reductions to MHF-SAT for prominent NP-complete problems, e.g. the Feedback Vertex Set, the Vertex Cover, the Dominating Set and the Hitting Set problem can easily be encoded as MHF.

Furthermore we consider restricted subclasses of MHF and show that they are NP-complete w.r.t. SAT.

Finally we provide algorithms for two NP-complete subclasses of MHF solving SAT significantly faster than in time $O((\sqrt[3]{3})^n) = O(1.443^n)$ the currently best worst case running time of an algorithm for solving unrestricted members of the class MHF [8]. The first of these subclasses consists of formulas, where the Horn part is negative monotone and the variable graph corresponding to the positive 2-CNF part P consists of disjoint triangles only. For this class we provide an algorithm and give the running times for the cases that H is k-uniform, for $k \in \{3, 4, 5, 6\}$ The motivation for looking at this subclass of MHF consists of the fact that the analysis of an algorithm for solving unrestricted MHF has its worst case behaviour just for this class of formulas.

The other NP-complete subclass of MHF actually consists of infinitely many subclasses with parameter $k \geq 3$. For fixed k a worst-case bound of $O(k^{\frac{n}{k}})$ is shown. For $k = 4, 5, 6, 10$ the bases of the exponential growth are $k^{\frac{1}{k}} \approx 1.415, 1.38, 1.259$ resp., going to 1 with k going to ∞. While the class looks artificial the derivation of the running time deserves attention. In this case enumerating minimal satisfying assignments of the Horn part of the input formulas helps, whereas for unrestricted MHF, and for the first subclass mentioned above enumerating minimal satisfying assignments of the 2-CNF part yields better bounds.

By the way, there is an interesting connection between MHF-SAT and unrestricted SAT presented in [8]: If there is some $\alpha < \frac{1}{2}$ such that each MHF $M = P \wedge H$, where P has $k \leq 2n$ variables, can be solved in time $O(\|M\|2^{\alpha k})$, then there is some $\beta \leq 2\alpha < 1$ such that SAT for an arbitrary CNF-formula F can be decided in time $O(\|F\|2^{\beta n})$. Here $||F||$ denotes the length of F. Although, there has been made some progress recently in finding non-trivial bounds for SAT for arbitrary CNF formulas [2,3], it would require a significant breakthrough in our understanding of SAT to obtain upper time bounds of the form $O(2^{(1-\epsilon)n})$, for some $\epsilon > 0$.

To fix the notation and basics, let CNF denote the set of duplicate-free conjunctive normal form formulas over propositional variables $x \in \{0, 1\}$. A *positive (negative)* literal is a (negated) variable. The *negation (complement)* of a literal l is $\bar{l}$. Each formula $F \in$ CNF is considered as a conjunction of clauses, and each clause $c \in F$ is a disjunction of literals, which in addition is assumed to be free of complemented pairs $\{x, \bar{x}\}$. For formula F, clause c, by $V(F), V(c)$ we denote the variables contained (neglecting negations), respectively. We demand

that clauses may not contain a literal more than once. The satisfiability problem (SAT) asks, whether input $F \in$ CNF has a *model*, which is a truth assignment $\alpha : V(F) \to \{0,1\}$ assigning at least one literal in each clause of F to 1. The truth assignment $\overline{\alpha}$ is obtained from α by complementing all assignments: $\overline{\alpha} := 1 - \alpha$. A *backbone variable* has the same value in each model of a satisfiable formula.

2 Classical NP-Complete Problems Encoded as MHF-SAT

In this section we provide reductions from some classical NP-complete problems to SAT and we will show that by complementing all literals of the corresponding SAT-instance F we obtain a mixed Horn instance $\overline{F}$. The Graph Colorability problem was mentioned already as an example of such a problem in [8]. But there are many more NP-complete problems which can easily be encoded into MHF-SAT. To demonstrate this, we explicitly transform four classical NP-complete problems, [4], to MHF-SAT, the Feedback Vertex Set, the Vertex Cover, the Hitting Set, and the Dominating Set problem. Inspecting the reductions of a dozen of further NP-complete problems to SAT, presented in [12], it turns out that most of them can easily be brought to the MHF-form.

The following abbreviations for propositional formulas are useful for our reduction: The formula *at_most_one* (*at_least_one*) defines that at most one (at least one) literal of the argument is true:

$$at\_most\_one\{l_1, \ldots, l_x\} := \bigwedge_{1 \le i < j \le x} (\overline{l_i} \vee \overline{l_j})$$
$$at\_least\_one\{l_1, \ldots, l_x\} := (l_1 \vee l_2 \vee \ldots \vee l_x)$$

With these fomulas it is easy to define *exactly_one*, which is true if, and only if, exactly one of its arguments is true:

$$exactly\_one\{l_1, \ldots, l_x\} := at\_most\_one\{l_1, \ldots, l_x\} \wedge at\_leat\_one\{l_1, \ldots, l_x\}$$

We begin with the **Feedback Vertex Set** Problem, which is defined as follows:

INSTANCE: $G = (V, A)$ a directed graph, $k \le |V|$ positive integer.
QUESTION: Is there a set $V' \subseteq V$, $|V'| \le k$, such that $G - \{V'\}$ has no directed circles?

The Feedback Vertex Set problem can be reduced to SAT as follows [12]:

$$X = \{[v,i] | v \in V, 1 \le i \le |V|\}$$
$$F = \bigwedge_{1 \le i \le |V|} exactly\_one\{[v,i] | v \in V\} \wedge \bigwedge_{v \in V} exactly\_one\{[v,i] | 1 \le i \le |V|\}$$
$$\wedge \bigwedge_{(u,v) \in A} \bigwedge_{k < i \le |V|} ([u,i] \Rightarrow at\_least\_one\{[v,j] | 1 \le j \le k, i < j \le |V|\})$$

This reduction is due to the fact, that any directed acyclic graph (DAG) admits a linear ordering of its vertices such that all arcs are directed from left to right. The first two parts $\bigwedge$ describe a one-to-one labeling of the vertices, where the vertices with labels $\leq k$ are considered to build V'. In the last part $\bigwedge\bigwedge$ the DAG property is checked for the reduced graph. The existence of an edge between a vertex u with a label $i > k$ and a vertex v with a label j, where $k < j < i$, is not allowed. Let α be a satisfying truth assignment of F. Then all the vertices $v \in V$, for which $\alpha([v,i]) = 1$ holds, where $1 \leq i \leq k$, belong to the feedback vertex set V' and vice versa. Now we will consider F and show that we can provide a reduction to a mixed Horn instance. Let $V = \{v_1, \ldots, v_n\}$ then:

$$F = \bigwedge_{1\leq i\leq n} \left(([v_1,i] \vee [v_2,i] \vee \ldots \vee [v_n,i]) \wedge \bigwedge_{1\leq j<l\leq n} (\overline{[v_j,i]} \vee \overline{[v_l,i]}) \right)$$
$$\wedge \bigwedge_{v\in V} \left(([v,1] \vee [v,2] \vee \ldots \vee [v,n]) \wedge \bigwedge_{1\leq j<l\leq n} (\overline{[v,j]} \vee \overline{[v,l]}) \right)$$
$$\wedge \bigwedge_{(u,v)\in A} \bigwedge_{k<i\leq n} \Big(\overline{[u,i]} \vee ([v,1] \vee [v,2] \vee \ldots \vee [v,k] \vee [v,i+1] \vee \ldots \vee [v,n]) \Big)$$

Complementing all literals in F we obtain $\overline{F} \in$ MHF:

$$\overline{F} = \bigwedge_{1\leq i\leq n} \bigwedge_{1\leq j<l\leq n} ([v_j,i] \vee [v_l,i]) \wedge \bigwedge_{v\in V} \bigwedge_{1\leq j<l\leq n} ([v,j] \vee [v,l])$$
$$\wedge \bigwedge_{1\leq i\leq n} (\overline{[v_1,i]} \vee \overline{[v_2,i]} \vee \ldots \vee \overline{[v_n,i]}) \wedge \bigwedge_{v\in V} (\overline{[v,1]} \vee \overline{[v,2]} \vee \ldots \vee \overline{[v,n]}) \wedge$$
$$\bigwedge_{(u,v)\in A} \bigwedge_{k<i\leq n} \Big([u,i] \vee (\overline{[v,1]} \vee \overline{[v,2]} \vee \ldots \vee \overline{[v,k]} \vee \overline{[v,i+1]} \vee \ldots \vee \overline{[v,n]}) \Big)$$

We have:

$$P := \bigwedge_{1\leq i\leq n} \bigwedge_{1\leq j<l\leq n} ([v_j,i] \vee [v_l,i]) \wedge \bigwedge_{v\in V} \bigwedge_{1\leq j<l\leq n} ([v,j] \vee [v,l])$$

and

$$H := \bigwedge_{1\leq i\leq n} (\overline{[v_1,i]} \vee \overline{[v_2,i]} \vee \ldots \vee \overline{[v_n,i]}) \wedge \bigwedge_{v\in V} (\overline{[v,1]} \vee \overline{[v,2]} \vee \ldots \vee \overline{[v,n]}) \wedge$$
$$\bigwedge_{(u,v)\in A} \bigwedge_{k<i\leq n} \Big([u,i] \vee (\overline{[v,1]} \vee \overline{[v,2]} \vee \ldots \vee \overline{[v,k]} \vee \overline{[v,i+1]} \vee \ldots \vee \overline{[v,n]}) \Big)$$

Obviously $\overline{F} = P \wedge H$ is a mixed Horn formula which satisfies: Let $\overline{\alpha}$ be a satisfying truth assignment of $\overline{F}$, then all the vertices $v \in V$ belong to the FVS V', for which holds: $\overline{\alpha}([v,i]) = 0$, where $1 \leq i \leq k$.

Similarly we can provide a reduction from the Feedback Arc Set problem to MHF-SAT. Recall that in the Feedback Arc Set problem instead of vertices we consider arcs.

Next we treat the Vertex Cover Problem, which is defined as follows:

INSTANCE: Graph $G = (V, E)$, positive integer $k \leq |V|$.
QUESTION: Is there a set $V' \subseteq V$, $|V'| \leq k$, such that for all $\{u, v\} \in E$ we have $\{u, v\} \cap V' \neq \emptyset$?

The Vertex Cover problem can be reduced to SAT as follows [12]:

$$X = \{[v, i] | v \in V, 1 \leq i \leq k\}$$
$$F = \bigwedge_{1 \leq i \leq k} at\_most\_one\{[v, i] | v \in V\}$$
$$\wedge \bigwedge_{(u,v) \in E} at\_least\_one\{[u, i], [v, i] | 1 \leq i \leq k\}$$

The first part $\bigwedge$ implies that at most k vertices are chosen (literals $[v, i]$ set to true), and the second part $\bigwedge$ verifies that the chosen vertex set is a vertex cover indeed.

We now consider F and after some calculation we will assert that a reduction to a mixed Horn instance is possible. Let $V = \{v_1, \ldots, v_n\}$ then we obtain:

$$F = \bigwedge_{1 \leq i \leq k} \bigwedge_{1 \leq j < l \leq n} (\overline{[v_j, i]} \vee \overline{[v_l, i]})$$
$$\wedge \bigwedge_{(u,v) \in E} ([u, 1] \vee [v, 1] \vee [u, 2] \vee [v, 2] \vee \ldots \vee [u, k] \vee [v, k])$$

Complementing all literals in F we obtain $\overline{F}$:

$$\overline{F} = \bigwedge_{1 \leq i \leq k} \bigwedge_{1 \leq j < l \leq n} ([v_j, i] \vee [v_l, i])$$
$$\wedge \bigwedge_{(u,v) \in E} (\overline{[u, 1]} \vee \overline{[v, 1]} \vee \overline{[u, 2]} \vee \overline{[v, 2]} \vee \ldots \vee \overline{[u, k]} \vee \overline{[v, k]})$$

For $\overline{F}$ we have: Let α be a satisfying truth assignment of $\overline{F}$, then all the vertices which belong to the literals $[v, i]$ assigned 0 to, form the vertex cover V'. We set

$$P = \bigwedge_{1 \leq i \leq k} \bigwedge_{1 \leq j < l \leq n} ([v_j, i] \vee [v_l, i])$$

and

$$H = \bigwedge_{(u,v) \in E} (\overline{[u, 1]} \vee \overline{[v, 1]} \vee \overline{[u, 2]} \vee \overline{[v, 2]} \vee \ldots \vee \overline{[u, k]} \vee \overline{[v, k]})$$

Obviously P is a positive monotone 2-CNF formula and H is a Horn formula. Thus the instance $\overline{F} = P \wedge H$ belongs to the class MHF.

The **Hitting Set** problem can also be encoded into MHF-SAT. It is defined as follows:

INSTANCE: Set C of subsets of a finite set S, a positive integer $k \leq |S|$.
QUESTION: Is there a set $S' \subseteq S$, $|S'| \leq k$, such that for all $c \in C$ we have $c \cap S' \neq \emptyset$?

The Hitting Set problem can be transformed to SAT as follows [12]:

$$X = \{[s,i] | s \in S, 1 \leq i \leq k\}$$
$$F = \bigwedge_{1 \leq i \leq k} at\_most\_one\{[s,i] | s \in S\} \wedge \bigwedge_{c \in C} at\_least\_one\{[s,i] | s \in c, 1 \leq i \leq k\}$$

This obviously is a generalization of the Vertex Cover Problem, where $|c| = 2$ for all $c \in C$. The first part choses at most k elements from S which will be part of S'. The second part verifies that this selection is indeed a hitting set. From F we obtain:

$$F = \bigwedge_{1 \leq i \leq k} \bigwedge_{1 \leq j < l \leq |S|} (\overline{[s_j,i]} \vee \overline{[s_l,i]}) \wedge \bigwedge_{c \in C} ([s_1,1] \vee \ldots \vee [s_1,k] \vee \ldots \vee [s_c,1] \vee \ldots \vee [s_c,k])$$

By $s_1, \ldots s_{|c|}$ we denote all elements of c, for an arbitrary $c \in C$. Complementing all literals of F we obtain a mixed Horn formula $\overline{F}$:

$$\overline{F} = \bigwedge_{1 \leq i \leq k} \bigwedge_{1 \leq j < l \leq |S|} ([s_j,i] \vee [s_l,i]) \wedge \bigwedge_{c \in C} (\overline{[s_1,1]} \vee \ldots \vee \overline{[s_1,k]} \vee \ldots \vee \overline{[s_c,1]} \vee \ldots \vee \overline{[s_c,k]})$$

It is obviously that $\overline{F} = P \wedge H$ a mixed Horn formula, where

$$P := \bigwedge_{1 \leq i \leq k} \bigwedge_{1 \leq j < l \leq |S|} ([s_j,i] \vee [s_l,i])$$

is a positive monotone 2-CNF formula and

$$H := \bigwedge_{c \in C} (\overline{[s_1,1]} \vee \ldots \vee \overline{[s_1,k]} \vee \ldots \vee \overline{[s_c,1]} \vee \ldots \vee \overline{[s_c,k]})$$

is a negative monotone Horn formula.

Next we consider the **Dominating Set Problem**, which is defined as follows:

INSTANCE: Graph $G = (V,E)$, a positive integer $k \leq |V|$.
QUESTION: Is there a set $V' \subseteq V$, $|V'| \leq k$, such that for all $v \in V$ we have $(\{v\} \cup N(v)) \cap V' \neq \emptyset$?

Transforming the Dominating Set problem to SAT can be done as follows [12]:

$$X = \{[v,i] | v \in V, 1 \leq i \leq k\}$$
$$F = \bigwedge_{1 \leq i \leq k} at\_most\_one\{[v,i] | v \in V\}$$
$$\wedge \bigwedge_{v \in V} at\_least\_one\{[w,i] | 1 \leq i \leq k, w \in (\{v\} \cup N(v))\}$$

The first $\bigwedge$ assures that at most k vertices are selected, and the second $\bigwedge$ verifies that the chosen vertex-set is a dominating set. We denote by $N(v)$ the set of all vertices which are adjacent to the vertex v. If F is satisfiable, then all the vertices belong to the dominating set which correspond to the variables assigned to 1. Let $V = \{v_1, \ldots, v_n\}$. Then we obtain from F:

$$\begin{aligned} F = & \bigwedge_{1\le i\le k} \bigwedge_{1\le j<l\le n} (\overline{[v_j,i]} \vee \overline{[v_l,i]}) \\ & \wedge \bigwedge_{v\in V} ([v,1] \vee \ldots \vee [v,k] \vee [v_{N_1},1] \vee \ldots \vee [v_{N_1},k] \vee [v_{N_2},1] \ldots \vee [v_{N_2},k] \vee \ldots \\ & \vee [v_{N_v},1] \ldots \vee [v_{N_v},k]) \end{aligned}$$

Let $v_{N_1}, \ldots, v_{N_v}$ be all the vertices adjacent to v in G. Complementing all literals of F we obtain $\overline{F}$:

$$\begin{aligned} \overline{F} = & \bigwedge_{1\le i\le k} \bigwedge_{1\le j<l\le n} ([v_j,i] \vee [v_l,i]) \\ & \wedge \bigwedge_{v\in V} (\overline{[v,1]} \vee \ldots \vee \overline{[v,k]} \vee \overline{[v_{N_1},1]} \vee \ldots \quad \vee \overline{[v_{N_1},k]} \vee \overline{[v_{N_2},1]} \ldots \vee \overline{[v_{N_2},k]} \vee \ldots \\ & \vee \overline{[v_{N_v},1]} \ldots \vee \overline{[v_{N_v},k]}) \end{aligned}$$

We set

$$P := \bigwedge_{1\le i\le k} \bigwedge_{1\le j<l\le n} ([v_j,i] \vee [v_l,i])$$

and

$$H := \bigwedge_{v\in V} (\overline{[v,1]} \vee \ldots \vee \overline{[v,k]} \vee \overline{[v_{N_1},1]} \vee \ldots \vee \overline{[v_{N_1},k]} \vee \overline{[v_{N_2},1]} \ldots \vee \overline{[v_{N_2},k]} \vee \ldots \vee \overline{[v_{N_v},1]} \ldots \vee \overline{[v_{N_v},k]})$$

Then P is a positive monotone 2-CNF formula and H is a negative monotone Horn formula. Thus $\overline{F} = P \wedge H$ is a mixed Horn formula. Considering any satisfying truth assignment of $\overline{F}$ all the vertices which correspond to the variables which are assigned 0 to belong to the dominating set.

In this way, we can show similarly that nearly all NP-complete problems introduced by Karp [4] have a natural and straightforward encoding as an MHF-SAT problem. Some of these problems are listed below and are treated in the full version of the paper [7]:

– The **Edge Dominating Set** Problem:

  INSTANCE: Graph $G = (V, E)$, a positive integer $k \le |E|$.
  FRAGE: Is there a set $E' \subseteq E$, $|E'| \ge k$, such that for all $e \in E$ we have that there exists a $f \in E'$, such that $e \cap f \neq \emptyset$?

– The **Idependent Set** Problem:

  INSTANCE: Graph $G = (V, E)$, a positive integer $k > 0$.
  QUESTION: Is there a set $V' \subseteq V$, $|V'| \ge k$, such that for all $\{u, v\} \in E$ we have $\{u, v\} \not\subseteq V'$?

– The **Minimum Maximal Matching** Problem:

INSTANCE: Graph $G = (V, E)$, a positive integer $k \leq |E|$.
QUESTION: Is there a set $E' \subseteq E$, $|E'| \leq k$, such that for all $\{u, v\} \in E$ we have $\{u, v\} \cap V(E') \neq \emptyset$?

The number of Boolean variables in the resulting member of MHF often is larger than the relevant instance size of the original problem, e.g. in case of the Feedback Vertex Set problem (FVS) n^2 vs. n. However, solving the MHF member by an up-to-date SAT solver is pretty fast, because of the large number of 2-clauses which allow for iterated unit resolution phases, in case of FVS of length n each time.

3 Some NP-Complete Subclasses of MHF

This section is devoted to establish NP-completeness of certain interesting subclasses of MHF. The first class to consider consists of formulas whose Horn clauses are k-uniform, that means they have all equal length k, where $k \geq 3$ and whose 2-CNF part is positive monotone. We denote this class by $\mathrm{MH_kF^+}$. Next we consider the class $\mathrm{MH_k^-F^+} \subset \mathrm{MH_kF^+}$ of mixed Horn formulas with a positive monotone 2-CNF part and negative monotone, k-uniform, Horn clauses, $k \geq 3$. Afterwards we consider a subclass of $\mathrm{MH_k^-F^+}$, for which additionally holds that the variable-graph G_P of the positive monotone 2-CNF part P consists of disjoint edges only. We denote such a class of formulas by $\mathrm{MH_k^-F^{d+}}$. Finally we consider the class $\mathrm{LMH_k^-F^+} \subset \mathrm{MH_k^-F^+}$ which consists of mixed Horn formulas of $\mathrm{MH_k^-F^+}$, for which additionally holds that the Horn clauses are linear [9]. A CNF formula F is called *linear* if

(1) F contains no pair of complementary unit clauses and
(2) for all $c_1, c_2 \in F : c_1 \neq c_2$ we have $|V(c_1) \cap V(c_2)| \leq 1$.

Theorem 1. SAT *is* NP*-complete for the following* MHF*-subclasses:* $\mathrm{MH_kF^+}$, $\mathrm{MH_k^-F^+}$, $\mathrm{MH_k^-F^{d+}}$ *and* $\mathrm{LMH_k^-F^+}$, *for* $k \geq 3$.

Proof. Note that the NP-completeness of all these MHF-subclasses directly follows from Schaefer's theorem [11], because none of these subclasses is properly contained in any tractable CNF class due to Schaefer's theorem. However, direct reductions are of interest per se. To that end, it is easy to see that the encoding of k-colorability for graphs, $k \geq 3$, to SAT directly yields the NP-completeness of $\mathrm{LMH_k^-F^+} \subset \mathrm{MH_k^-F^+} \subset \mathrm{MH_kF^+}$. But this connection to graph colorability does not hold for the class $\mathrm{MH_k^-F^{d+}}$, for which we provide the following reduction: It is well-known that the class k-CNF, $k \geq 3$, is NP-complete. Let $F \in k\text{-CNF}$ be an arbitrary formula. Then we can reduce F to a SAT-equivalent formula $\widetilde{M_F}$ of the class $\mathrm{MH_k^-F^{d+}}$ in polynomial time. The transformation consists of two main steps, the first of which is refered to as MHF-*reduction*:
1. Let $V^+(F) \subseteq V(F)$ be the set of all variables having a positive occurrence in F. For each variable $x \in V^+(F)$ we introduce a new variable $y_x \notin V(F)$ and

perform the following steps: We replace each positive occurrence of $x \in V^+(F)$ in the k-clauses by $\overline{y_x}$, for each $x \in V^+(F)$. Let F' be the resulting formula. Next we add the constraints $\overline{y_x} \Leftrightarrow x$, for all $x \in V^+(F)$, to F', equivalent to

$$(\overline{y_x} \Leftrightarrow x) \Leftrightarrow ((y_x \vee x) \wedge (\overline{y_x} \vee \overline{x}))$$

yielding the new formula

$$M_F = F' \wedge \bigwedge_{x \in V^+(F)} (y_x \vee x) \wedge (\overline{y_x} \vee \overline{x})$$

Here F' only consists of Horn clauses of length k.

2. As formulas of each class $\mathrm{MH}_{\mathrm{k}}^{-}\mathrm{F}^{\mathrm{d}+}$ are allowed to contain positive 2-clauses only, but M_F also contains the negative monotone 2-clauses $(\overline{y_x} \vee \overline{x})$, for each $x \in V^+(F)$, we add to each such negative 2-clause exactly $(k-2)$ backbone variables z_x^i and the formulas $F_x^i \in \mathrm{MH}_{\mathrm{k}}^{-}\mathrm{F}^{\mathrm{d}+}$,for $i = 1, \ldots, k-2$, such that z_x^i is a backbone variable of F_x^i which has to be set to 1. All such formulas F_x^i, $i = 1, \ldots, k-2$, must be pairwise and also with F' variable-disjoint. An example of such a backbone-formula is:

Let $i = 1, \ldots k-2$.

$$F_x^i = (z_{x1}^i \vee z_{x2}^i) \wedge (z_{x3}^i \vee z_{x4}^i) \wedge (z_{x5}^i \vee z_{x6}^i) \wedge \ldots \wedge (z_{x(2k-1)}^i \vee z_{x(2)k}^i)$$
$$\wedge (\overline{z_{x1}^i} \vee \overline{z_{x3}^i} \vee \ldots \vee \overline{z_{x(2k-1)}^i}) \wedge \ldots \wedge (\overline{z_{x2}^i} \vee \overline{z_{x4}^i} \vee \ldots \vee \overline{z_{x(2k-1)}^i})$$

The Horn part consists of 2^{k-1} negative monotone k-clauses, where each k-clause consists of the variables of a vertex cover of the positive monotone 2-CNF part $(z_1^i \vee z_2^i) \wedge (z_3^i \vee z_4^i) \wedge (z_5^i \vee z_6^i) \wedge \ldots \wedge (z_{2k-1}^i \vee z_{2k}^i)$. Since the positive monotone 2-CNF part has altogether 2^k negative monotone vertex covers, but there are only 2^{k-1} negative monotone k-clauses in the Horn part, the formula F_x^i is satisfiable and has k backbone-variables, which have to be set to 1. We assume that the vertex cover which selects from each positive 2-clause the variable with the odd index does not appear as negative Horn clause in F_x^i. Thus each variable of this clause is a backbone variable of F_x^i.

Finally we add to each negative monotone clause $(\overline{y_x} \vee \overline{x})$ of M_F the literals $\overline{z_x^1}, \ldots, \overline{z_x^{k-2}}$, for $x \in V^+(F)$, and obtain $(\overline{y_x} \vee \overline{x} \vee \overline{z_x^1} \vee \ldots \vee \overline{z_x^{k-2}})$. Altogether we obtain

$$\widetilde{M_F} = F' \wedge \bigwedge_{x \in V^+(F)} \left((\overline{y_x} \vee \overline{x} \vee \overline{z_x^1} \vee \ldots \vee \overline{z_x^{k-2}}) \wedge F_x^1 \wedge \ldots \wedge F_x^{k-2} \right)$$
$$\wedge \bigwedge_{x \in V^+(F)} (y_x \vee x) \in \mathrm{MH}_{\mathrm{k}}^{-}\mathrm{F}^{\mathrm{d}+}$$

It holds that $\widetilde{M_F}$ is satisfiable if and only if F is satisfiable: We assume that F is satisfiable. Let α be a model of F, then we set all the variables, which $\widetilde{M_F}$ and F have in common, also according to α. Each newly introduced variable y_x

equivalent to $x \in V(F)$ is set as follows: $y_x = 1 - \alpha(x)$. Since the added backbone formulas F_x^i are always satisfiable, $\widetilde{M_F}$ is also satisfiable. If F is unsatisfiable, then we obviously cannot satisfy F' and hence $\widetilde{M_F}$. □

4 Algorithms for SAT of Further Mixed Horn Classes

In this section we consider some special classes of mixed Horn formulas, for which we can solve SAT in a running time better than $O(2^{0.5284n})$. Let H be a k-uniform ($k \geq 3$), negative monotone and linear formula, with the following properties: All clauses can be arranged in $c_1, \ldots, c_{|H|}$ and for each clause c_i all the literals can be arranged such that either the last literal of the clause c_i ($i = 1 \ldots, |H|$) and the first literal of c_{i+1} are equal or the clauses are variable-disjoint. Further, each clause c_i, $i \in \{1, \ldots, |H|\}$ is not allowed to share a variable with any other clause except with c_{i-1} and c_{i+1}. Then we say H has *overlappings in boundary variables* only.

Definition 1. *We consider formulas $M = G \wedge H \in$ MHF with the following properties: G consists of 2-clauses not necessarily positive monotone and H consists of linear Horn clauses, for which holds: All clauses are negative monotone, k-uniform, $k \geq 3$, and there is an ordering $c_1, c_2 \ldots, c_{|H|}$ of the clauses of H and an ordering of all literals in each clause, such that H has overlappings in boundary variables only. We denote this class by* k-BLMHF (k-Boundary-Linear mixed Horn formulas).

Each connected component of the incidence graph G_H for H, looks like Figure 1. Note that H with this restriction belongs to a subclass of the class of nested formulas studied by Knuth [6].

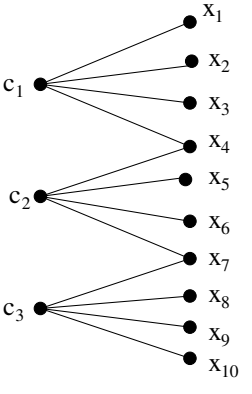


Fig. 1. The incidence graph for $H = (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3} \vee \overline{x_4}) \wedge (\overline{x_4} \vee \overline{x_5} \vee \overline{x_6} \vee \overline{x_7}) \wedge (\overline{x_7} \vee \overline{x_8} \vee \overline{x_9} \vee \overline{x_{10}})$

Theorem 2. SAT *remains* NP*-complete for the class* k-BLMHF*,* $k \geq 3$.

Proof. We provide a polynomial time reduction from k-CNF-SAT, which is NP-complete to k-BLMHF-SAT, proving the NP-completeness of the latter. Let $F \in k$-CNF. First we perform the MHF-reduction step as in the proof of Theorem 1 obtaining the corresponding formula M_F. Let the clauses of F' be labeled as follows: $c_1, c_2, \ldots, c_{|F'|}$. Then we proceed as follows:

At the beginning let $i = 1$, as long as $i < |F'|$: We consider c_i. As long as c_i has a common variable x with c_j, for some $j \geq i+1$, we replace $\overline{x}$ in c_j by $\overline{y_x}$, y_x not yet occurring in the set of variables of M_F, and add the 2-clauses $(\overline{x} \vee y_x) \wedge (\overline{y_x} \vee x)$ to M_C. Now we set $i := i + 1$.

Obviously M_F equivalent to F concerning SAT and the transformation can be performed in polynomial time. □

Theorem 3. SAT *for* k-BLMHF*,* $k \geq 3$*, can be solved in time* $O((\sqrt[k]{k})^n)$.

Proof. The following algorithm solves SAT for the class k-BLMHF in polynomial time:

Algorithm k-BLMHF

INPUT: $F \in k$-BLMHF, $F = G \wedge H$, where G consists of 2-clauses and H is k-uniform, negative monotone, linear and has overlappings in boundary variables only.

OUTPUT: A satisfying truth assignment for F, if F is satisfiable. Else F is not satisfiable.

begin
1. Compute all minimal Hitting Sets of H, let these be S_i, $i = 1, \ldots, m$.
2. Set $i = 1$.
3. As long as no satisfying truth assignment is found and $i \leq m$, do:

(a) Assign 0 to all variables of S_i and 1 to all other variables and check, whether this assignment satisfies G.
(b) If it does, then the assignment is a truth assignment for F and the algorithm stopps.
(c) If not, then check for S_{i+1}.

4. Return F is unsatisfiable.

end

Correctness. The algorithm k-BLMHF verifies for each minimal hitting set of H, whether the partial truth assignment setting all variables in the hitting set to 0 can be extended to a model of F by checking the remaining 2-CNF part in linear time [1]. Since we consider only the minimal hitting sets of H we do not perform any restriction concerning the satisfiability of H, because for each model of F the set of all variables, which are set to 0 either corresponds to a minimal hitting set of H or it contains a minimal hitting set of H itself.

Analysis of the running time. Let $F \in k$-BLMHF with n variables. One can show that the number of minimal hitting sets is maximal if the Horn part of F consists of disjoint clauses only. So the running time of the algorithm k-BLMHF is dominated by this subclass of k-BLMHF, for which the number of minimal hitting sets of H is $k^{\lceil n/k \rceil}$ yielding the running time $O((\sqrt[k]{k})^n)$. As the sequence $(\sqrt[k]{k})_{k\in\mathbb{N}}$ decreases monotonically with increasing k, we obtain for $k \geq 4$ a running time better than $3^{n/3}$. □

The next NP-complete subclass of MHF we consider is the class MH_F$^{\triangle}$ consisting of mixed Horn formulas $M = P \wedge H$, with a negative monotone Horn part H and a positive monotone 2-CNF part P, for which holds that the corresponding variable graph G_P consists of disjoint triangles only. We further assume that $V(P) = V(H)$.

Theorem 4. SAT *remains* NP*-complete for the class* MH$_3^-$F$^{\triangle}$.

Proof. In Theorem 1 is shown, that SAT remains NP-complete for the class MH$_k^-$F^{d+}. Now we provide a polynomial time reduction from MH$_k^-$F^{d+}-SAT to MH_F$^{\triangle}$-SAT. Let $F \in$ MH$_k^-$F^{d+} with a Horn part consisting of negative monotone clauses and with a positive monotone 2-CNF part P, whose corresponding variables graph G_P consists of disjoint edges only:

$$(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge \ldots$$

Then for each such $(x_i \vee x_{i+1})$ we introduce a new variable $y_{i,i+1}$ and enlarge the P part to $\widetilde{P}$:

$$\begin{aligned}&(x_1 \vee x_2) \wedge (x_1 \vee y_{1,2}) \wedge (x_2 \vee y_{1,2}) \wedge (x_3 \vee x_4) \wedge (x_3 \vee y_{3,4}) \wedge (x_4 \vee y_{3,4})\\ &\wedge (x_5 \vee x_6) \wedge (x_5 \vee y_{5,6}) \wedge (x_6 \vee y_{5,6}) \wedge \ldots\end{aligned}$$

Now obviously $G_{\widetilde{P}}$ consists of disjoint triangles. Further we add to H the following clauses each one consisting of all variables of the same triangle of P, but all negated: $(\overline{x_1} \vee \overline{x_2} \vee \overline{y_{1,2}}) \wedge (\overline{x_3} \vee \overline{x_4} \vee \overline{y_{3,4}}) \wedge (\overline{x_5} \vee \overline{x_6} \vee \overline{y_{5,6}}) \wedge \ldots$ and obtain $\widetilde{H}$. Setting $y_{i,i+1}$ to 1, for all i we can satisfy the new added 2-clauses and also the new added Horn clauses. Thus $\widetilde{F} = \widetilde{P} \wedge \widetilde{H}$ is satisfiable if and only if F is satisfiable. □

Now we present an algorithm solving SAT for formulas in MH_F$^{\triangle}$. The running time is stated below for k-uniform Horn parts, for $k \in \{3, 4, 5, 6\}$. The proof of the running time is rather technical, therefore it is omitted and is contained in the full version of the paper [7]. By backtracking variables of the Horn part are set to 0 without violating P. To that end, in each triangle at least two variables must be set to 1. When setting a variable x to 0 to satisfy a clause in the Horn part, the other two variables of the triangle $\triangle_x$ must be set to 1. Clauses of H that are satisfied are put on a stack. Variables set to 0 are stored in the set V and those set to 1 are stored in the set W.

ALGORITHM MH_F$^{\triangle}$

INPUT: Let a formula $M = (P \wedge H) \in$ MH_F$^{\triangle}$ be given and let k be the number of triangles in G_P.
OUTPUT: M **is satisfiables**, if there is a satisfying truth assignment for M. Else: M **is not satisfiable**.

begin
1.) Let $H = c_1 \wedge c_2 \wedge ... \wedge c_h$, where $|c_1| \leq \ldots \leq |c_h|$.
2.) Delete all clauses c_j, for which there is a clause c_i, $(i < j)$, such that $c_i \subset c_j$.
3.) Delete all clauses c_i of H, where $V(\triangle_j) \subset V(c_i)$, for a $\triangle_j \in G_P$.
4.) Set $V := \emptyset$, $W := \emptyset$, $i := 1$ and $s_H \leftarrow create$.
5.) For all $i = 1, \ldots, h$ set a Pointer $*$ in front of the first literal in the clause c_i:

$$(c_i, *) = (*\overline{x_{i_1}} \vee \ldots \vee \overline{x_{i_m}})$$

6.) Perform **Procedure Backtrack(c_1,$*$)**.

end

Procedure Backtrack(c_i,$*$)
1.) Set all variables of the unit clauses and save all variables assigned 0 in V and all variables assigned 1 in W. When assigning 0 to a variable x search for the triangle $\triangle_x \in G_P$ also containing x and set the two other variables of $\triangle_x$ to 1. If there are complementary unit clauses or another contradiction, then the procedure stopps and returns M **is not satisfiable**.
2.) If $(c_i, *) = (\overline{x_{i_1}} \vee \ldots \vee \overline{x_{i_l}} \vee \ldots \vee \overline{x_{i_m}}*)$ and $i = 1$, then the procedure stopps end returns M **is unsatisfiable**.
3.) If $(c_i, *) = (\overline{x_{i_1}} \vee \ldots \vee \overline{x_{i_l}} \vee \ldots \vee \overline{x_{i_m}}*)$ and $i \neq 1$, then perform the following steps:

(a) Perform the operation pop_{s_H}: Delete all the clauses from the stack, which have been put on the stack in the last procedure call, and insert them in the original order into H.
(b) Delete all variables x_i saved in V in the last procedure call from V and save them in W, after deleting the two other variables from $\triangle_{x_i}$ from W.
(c) Verify, if there are two variabels of the same triangle saved in W. If yes, then set the third variable of this triangle to 0 and save it in $\widetilde{V}$.
(d) Set the Pointer at the beginning of the clause c_i:

$$c_i := (*\overline{x_{i_1}} \vee \ldots \vee \overline{x_{i_l}} \vee \ldots \vee \overline{x_{i_m}})$$

(e) Call the **Procedure Backtrack(c_{i-1},$*$)** .

4.) If $(c_i, *) = (\overline{x_{i_1}} \vee \ldots \vee \ *\overline{x_{i_l}} \vee \ldots \vee \overline{x_{i_m}})$, then set $j := i_l$.
5.) As long as $x_j \in W$ and $j \leq i_m$, augment j: $j = j + 1$.
6.) If $j = i_m + 1$, then: Perform the same steps as in 3. a)-e).
7.) If $x_j \notin W$ and $j \leq i_m$, then perform the following steps:

a) Set $x_j := 0; V := V \cup \{x_j\}; W := W \cup \triangle_{x_j}$;
b) Set the Pointer in front of $\overline{x_{j+1}}$:

$$(c_i, *) := (\overline{x_{i_1}} \vee \ldots \vee \overline{x_j} \vee *\overline{x_{j+1}} \vee \ldots \vee \overline{x_{i_m}})$$

c) Delete all clauses from H, which contain $\overline{x_j}$ and put them on stack s_H: $s_H := push_{s_H}(C_{x_j})$, where C_{x_j} is the set of all remaining Horn clauses containing the variable x_j.
d) If H is empty, then the procedure stops and returns M **is satisfiable**.
e) If H is not empty, then call the **Procedure Backtrack(c_{i+1},*)**.

Note that the algorithm works for arbitrary negative Horn parts. Specifically for k-uniform Horn parts, a careful analysis of yields the following results:

Theorem 5. *[7] Algorithm* MH_F$^{\triangle}$ *decides satisfiability for* $M \in$ MH_F$^{\triangle}$ *over* n *variables in time* $O(1.336^n)$*, for* $k = 3$*, in* $O(1.384^n)$*, for* $k = 4$*, in* $O(1.397^n)$*, for* $k = 5$ *and in* $O(1.408^n)$*, for* $k = 6$.

For $k = 3$ the running time is better than the so far best runnig time $O(1.427^n)$, [5], for mixed Horn formulas with an arbitrary 3-uniform Horn part.

Remark 1. *For* $k = 3$ *a better running time of* $O(1.273^n)$ *can be achieved using a clever branching strategy. However, it is not obvious how to extend this branching algorithm to the cases* $k \geq 4$.

Acknowledgement. We want to thank the anonymous referees for their valuable comments, especially the first reviewer for establishing Remark 1.

References

1. Aspvall, B., Plass, M.R., Tarjan, R.E.: A linear-time algorithm for testing the truth of certain quantified Boolean formulas. Inform. Process. Lett. 8, 121–123 (1979)
2. Dantsin, E., Wolpert, A.: Algorithms for SAT based on search in Hamming balls, ECCC Report No. 17 (2004)
3. Dantsin, E., Wolpert, A.: A faster clause-shortening algorithm for SAT with no restriction on clause length. J. Satisfiability, Boolean Modeling and Computation 1, 49–60 (2005)
4. Karp, R.M.: Reducibility Among Combinatorial Problems. In Complexity of Computer Computations. In: Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights, pp. 85–103. Plenum, New York (1972)
5. Kottler, S., Kaufmann, M., Sinz, C.: A New Bound for an NP-Hard Subclass of 3-SAT Using Backdoors. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 161–167. Springer, Heidelberg (2008)
6. Knuth, D.E.: Nested Satisfiability. Acta Informatica 28, 1–6 (1990)
7. Porschen, S., Schmidt, T., Speckenmeyer, E.: MHF – a central subclass of CNF, working paper, Universität zu Köln (2009)
8. Porschen, S., Speckenmeyer, E.: Satisfiability of Mixed Horn Formulas. Discrete Appl. Math. 155, 1408–1419 (2007)
9. Porschen, S., Speckenmeyer, E., Zhao, X.: Linear CNF formulas and satisfiability. Discrete Appl. Math. 157, 1046–1068 (2009)

10. Randerath, B., Speckenmeyer, E., Boros, E., Hammer, P., Kogan, A., Makino, K., Simeone, B., Cepek, O.: A Satisfiability Formulation of Problems on Level Graphs. In: ENDM, vol. 9 (2001)
11. Schaefer, T.J.: The complexity of satisfiability problems. In: Proc. STOC 1978, pp. 216–226. ACM Press, New York (1978)
12. Stamm-Wilbrandt, H.: Programming in propositional Logic or Reductions: Back to the Roots (Satisfiability), Technical Report, Universität Bonn (1991)

Variable Influences in Conjunctive Normal Forms

Patrick Traxler*

Institute of Theoretical Computer Science, ETH Zürich, Switzerland
patrick.traxler@inf.ethz.ch

Abstract. We provide an upper bound on the total influence of Boolean functions defined by k-cnfs. Our bound is nearly optimal. We achieve it by an extension and appropriate use of an algorithm of Paturi, Pudlák, and Zane. We also discuss applications to prove and compute lower bounds for the maximum clause width k.

1 Introduction

Changing the value of a random point $x \sim \mu$, μ a distribution, in coordinate i can change the value $f(x)$ of a Boolean function $f : \{0,1\}^n \to \{0,1\}$. The probability that the value changes is called *variable influence* and denoted by $\mathcal{I}_i^\mu(f)$. The *total influence* is the sum over all variable influences and denoted by $\mathcal{I}^\mu(f)$. It is one of the key concepts in the analysis of Boolean functions and related to concepts and problems in different areas. A by far not complete list includes constant depth circuit complexity, the study of voting systems, existence of sharp thresholds in random structures. See also [14,19].

We provide a nearly tight upper bound on the total influence of k-cnfs. The bound fulfills an optimality criterion introduced by Kahn & Kalai [12] which we will discuss below. The benefit of our bound is that it allows lower bounds for the maximum clauses width of Boolean functions with a sparse amount true points. We also discuss the problem of computing variable influences.

1.1 Upper Bounds on Total Influence and Applications

We state our result first. Notation is introduced in Sec. 2. Define

$$u_\mu := \max_{i=1}^{n}\{(\mu_i \log(1/\mu_i))^{-1}, ((1-\mu_i)\log(1/(1-\mu_i)))^{-1}\}$$

for a product distribution $\mu = (\mu_1, ..., \mu_n)$. Define $\mu(f) := \Pr_{x\sim\mu}(f(x) = 1)$.

Theorem 1. *Let μ be a product distribution. For every k-cnf function f with $\mu(f) > 0$,*

$$\mathcal{I}^\mu(f) \leq u_\mu \, k \, \mu(f) \, \log(1/\mu(f)).$$

* This work was supported by the Swiss National Science Foundation SNF under project 200021-118001/1.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 101–113, 2009.

A weaker bound was discovered by Boppana in the context of circuit complexity. Boppana notes in [4] that $\mathcal{I}^{\mu}(f) \leq 2k$ if f is a k-cnf or k-dnf and μ the uniform distribution. Boppana uses this fact together with Håstad's Switching Lemma to show $\mathcal{I}^{\mu}(f) \leq 2k\,\lceil 10\log(4s)\rceil^{d-2}$ for any f computable by a circuit of bottom fan in at most k, depth d, and size s. This result improves on Linial et al. [16]. A further improvement was obtained by Håstad [9]. A standard application uses the fact that parity has total influence n w.r.t. the uniform distribution. We conclude from it that parity and actually any Boolean function with large total influence requires exponentially large circuits of constant depth.

Our bound is stronger in the following sense. It implies $\mathcal{I}^{\mu}(f) \leq 1.062\,k$, μ the uniform distribution, and it supports our intuition that the total influence should be small if $\mu(f)$ is small. We will discuss the optimality of this bound below and also how it relates to work of Kahn & Kalai [12]. The benefit of our bound is that is non-trivial for sparse f, i.e., f with $\mu(f) = 2^{cn}$ for some $c < 1$ which depends on k. The trivial upper bound for $\mathcal{I}(f)$ is $2\,n\,\mu_{1/2}(f)$. It is better than our bound as soon as $\mu_{1/2}(f) < 2^{-n/k}$. Another benefit is that the bound holds for the Bernoulli distribution which we denote by μ_p. The Bernoulli distribution is used for example in the random graph model in which every edge is chosen with probability p. A graph property like connectivity induces a Boolean function and our bound applies to such functions, i.e., random graph properties.

Our upper bound can be used to prove or compute lower bounds for the maximum clause width. The maximum clause width is one of the critical parameters in SAT-solving. Small clause width seems to be necessary for fast SAT-solving of general cnfs. Impagliazzo & Paturi [11] give a theoretical justification for this. Their result implies that we can not reduce efficiently the maximum clause width to, say, 3 with only introducing $o(n)$ many new variables, unless we can solve k-SAT in subexponential time. With regard to this observations we consider the following problem. We have given a Boolean function $f : \{0,1\}^n \to \{0,1\}$ and we want to know a lower bound for the smallest k such that f is definable by a k-cnf.

There are different ways to prove lower bounds for the total influence of a Boolean function. For example, Kahn et al. [13] prove that any Boolean function has a highly influential variable. This implies that any symmetric Boolean function has high total influence. Nisan & Szegedy [18] relate the total influence of a Boolean function to the maximum degree of a polynomial defining it. O'Donnell et al. [20] show that Boolean functions of low decision tree complexity have a highly influential variable.

Computing a lower bound for the maximum clause width is probable more interesting from a practical point of view. Despite computational results in learning theory, researchers in computer science and artificial intelligence started recently to consider computing variables influences. Considered function classes are for example given by the weighted majority rule [17,15], network flow games or (s,t)-connectivity games [2]. The problem is usually #P-complete [23,17,2]. Bachrach et al. [1] provide an algorithm for approximating variable influences up to an

additive error $\pm\varepsilon$ of any Boolean function using only membership queries. We discuss in Sec. 4 how to use SAT-solvers to compute variable influences.

1.2 Related Work: PPZ and Its Extension

The proof of the upper bound is probabilistic. It has its origin in circuit complexity. Paturi et al. [22] used it originally to prove that parity requires OR-AND-OR circuits, so called Σ_3-circuits, with at least $\Omega(n^{1/4}\,2^{\sqrt{n}})$ many gates. This bound is tight up to a constant factor. They generalized their result and showed that $\mathcal{I}^{\mu}(f) = O(\log(sn)^2)$, s the number of gates in a Σ_3-circuit computing f, and μ the uniform distribution. Their bound works however only if $\mu(f) = \frac{1}{2}$. They also observe that their technique yields a randomized algorithm of running time roughly $O(2^{(1-1/k)n})$ to decide satisfiability of a given k-cnf. Calabro et al. [5] used the Edge-Isoperimetric Inequality to show that the algorithm is sensitive to the number of satisfying assignments. The edge-isoperimetry of a Boolean function is closely related to its total influence.

Our contributions are an extension of the analysis in [22,5] to an algorithm in which every variable is set with a given probability. The probabilities may differ. Moreover, we use the analysis to bound the total influence. We use the observation that the success probability of the algorithm is high if the total influence is large. This implies an upper on the total influence since the success probability can not be too large. This observation also allows us to conclude Theorem 1 for the uniform case directly from the analysis of PPZ [22,5] if we additionally observe that the degree of isolation $I(x)$ of a satisfying assignment x, as used in [22,5], and the total influence are closely related. More precisely, let μ be the uniform distribution. Then, $\mathcal{I}^{\mu}(f) = 2^{-n}\,(\sum_{x:f(x)=1} n - I(x))$.

1.3 Background: Optimal Low Total Influence

Kahn & Kalai [12] raise several conjectures about "optimal" functions in [12]. Any of these conjectures is still unverified to our knowledge but would have implications for the study of k-cnf functions as well. This is what will point next.

Definition 2 (Kahn & Kalai [12]). *Let $C \geq 0$ and $0 < p < 1$ and μ_p denote the Bernoulli distribution with bias p. A (monotone increasing) Boolean function f with $\mu_p(f) > 0$ is called* (C,p)-optimal *iff*

$$\mathcal{I}^{\mu_p}(f) \leq \frac{C}{p\log(1/p)}\,\mu_p(f)\,\log(1/\mu_p(f)).$$

The reason for Kahn & Kalai to name their concept (C,p)-optimal comes from the Edge-Isoperimetric Inequality and a generalization of it to the Bernoulli distribution [12] based on Log-Sobolev Inequalities. These inequalities show that the total variable influence of (C,p)-optimal functions is optimal up to the factor C and a factor depending on p. Kahn & Kalai [12] showed

$$\mathcal{I}^{\mu_p}(f) \geq \frac{1}{p\log(1/p)}\,\mu_p(f)\,\log(1/\mu_p(f))$$

for any monotone increasing f. (The Edge-Isoperimetric Inequality corresponds to the case $p = 1/2$ and f an arbitrary Boolean function.) An implication of Theorem 1 is.

Corollary 3. *Let $0 < p \leq \frac{1}{2}$. Every k-cnf function is (k, p)-optimal.*

We determined therefore the total variable influence of k-cnf functions up to a factor depending on k and p and also identified a first important class of (C, p)-optimal functions. For this special case and $p = \frac{1}{2}$ we can show that Conjecture (6a) from [12] holds.

Conjecture 4 (Kahn & Kalai [12]). Given C, there are $K, \delta > 0$ such that for any $(C \log(1/p), p)$-optimal and monotone increasing function f with $\mu_p(f) > 0$ there exists $I \subseteq \{1, ..., n\}$ of size at most $K \log(1/\mu_p(f))$, such that $\mu_p(f(x) = 1 \mid x_i = 1 \; \forall i \in I) \geq (1 + \delta)\, \mu_p(f)$.

Theorem 5. *Let f be a monotone increasing k-cnf function. There exists $I \subseteq \{1, ..., n\}$ of size at most $O(k\, 2^k \log(1 + \delta))$ such that*

$$\Pr_{x \sim \mu_{1/2}} (f(x) = 1 \mid x_i = 1 \; \forall i \in I) \geq (1 + \delta)\, \mu_{1/2}(f).$$

Similar results for the uniform distribution and k-cnfs are also known. Theorem (3.5) in [10] and Lemma (7) from [24]. We will prove the theorem actually for any (not necessarily monotone) k-cnf function and the index set I can be any set which contains all the variable indices of enough clauses.

Not every C-optimal function is a C-cnf function. O'Donnell & Wimmer [21] provide a (monotone) function f with $\mathcal{I}(f) = O(\log(n))$ such that any cnf which agrees on at least a 0.1-fraction of the points with f has size at least $2^{\Omega(n/\log(n))}$. Henceforth, even approximate representations are impossible.

2 Notation

2.1 Variable Influences and Expectations

We assume that $0 < \mu_i < 1$ for every $i \in [n] = \{1, ..., n\}$. A *product distribution* μ over $\{0,1\}^n$ is given by $(\mu_1, ..., \mu_n)$ and the probability mass function $\mu(x) := \prod_{i=1}^n \mu_i^{x_i} (1 - \mu_i)^{1-x_i}$, $x \in \{0,1\}^n$. Let e_i be the point with the i-th bit 1 and 0 elsewhere and let $\oplus$ denote addition modulo 2. The *variable influence* of the i-th variable is defined as

$$\mathcal{I}_i^{\mu}(f) := \Pr_{x \sim \mu} (f(x) \neq f(x \oplus e_i)).$$

The *total influence* is defined as $\mathcal{I}^{\mu}(f) := \sum_{i=1}^n \mathcal{I}_i^{\mu}(f)$. The expectation $\mathbb{E}_{\mu}[\,]$ is defined w.r.t. μ and for functions $h : \{0,1\}^n \to \mathbb{R}$.

$$\mathbb{E}_{\mu}[h] := \sum_{x \in \{0,1\}^n} h(x)\, \mu(x).$$

The quantity $\mu(f) := \Pr_{x\sim\mu}(f(x) = 1)$ is sometimes called the expectation of f since $\mu(f) = \mathbb{E}_\mu(f)$ for $\{0,1\}$-valued f. We refer to it as the *bias* of f.

We make the following conventions. We write μ_p for the Bernoulli distribution, i.e., $\mu_{p,i} = p$ for every $i \in [n]$. Whenever we omit μ we assume the uniform distribution. For example, if $f(x_1, ..., x_n) = 1 \oplus \bigoplus_{i=1}^n x_i$ then $\mu_{1/2}(f) = \frac{1}{2}$ and $\mathcal{I}(f) = n$.

2.2 Conjunctive Normal Forms

A *k-cnf function* f is a Boolean function $f : \{0,1\}^n \to \{0,1\}$ for which there exist $P_1, ..., P_m \subseteq [n]$ and $N_1, ..., N_m \subseteq [n]$ such that $|P_i| + |N_i| \leq k$ for every i and

$$f(x_1, ..., x_n) = \bigwedge_{i=1}^{m} \bigvee_{j\in P_i} x_j \vee \bigvee_{k\in N_i} \overline{x}_k.$$

The ORs in this definition are called *clauses* and the whole expression a *k-cnf*. By exchanging $\wedge$ and $\vee$ we get the dual expression which is called a *k-dnf*. The ANDs in a k-dnf are called *terms*. We call a point which maps to 1 (resp. 0) under f a *true point* (resp. *false point*) of f. Every true point of a k-cnf function is a satisfying assignment of a corresponding k-cnf and vice versa.

3 Upper Bound on Total Influence

We will define a distribution D on $\{0,1\}^n$ which is strongly concentrated on the true points of a k-cnf function if its total influence is large. Using standard inequalities from probability, like Jensen's inequality, we conclude that the concentration and therefore the total influence can not be too large.

Some Preparation. Let $x \in \{0,1\}^n$, $i \in [n]$, and $j \in \{0,1\}$. Define $s_j(f, x, i)$ to be 1 if $x_i = j$ and $f(x) \neq f(x \oplus e_i)$, otherwise 0. The identity $\mathcal{I}_i^\mu(f) = \mathbb{E}_{x\sim\mu}[s_0(f, x, i) + s_1(f, x, i)]$ is immediate. We will need however the following.

Lemma 6. *It holds that*

$$\mathcal{I}_i^\mu(f) = \frac{1}{1-\mu_i}\,\mathbb{E}_\mu[f(x)\, s_0(f, x, i)] + \frac{1}{\mu_i}\,\mathbb{E}_\mu[f(x)\, s_1(f, x, i)]. \tag{1}$$

Proof. To prove the claim it suffices to show that

$$\frac{\mu_i}{1-\mu_i}\,\mathbb{E}_\mu[f(x)\, s_0(f, x, i)] = \mathbb{E}_\mu[(1 - f(x))\, s_1(f, x, i)] \tag{2}$$

and

$$\frac{1-\mu_i}{\mu_i}\,\mathbb{E}_\mu[f(x)\, s_1(f, x, i)] = \mathbb{E}_\mu[(1 - f(x))\, s_0(f, x, i)] \tag{3}$$

since

$$\mathcal{I}_i^{\mu}(f) = \mathbb{E}_{\mu}[|f(x) - f(x \oplus e_i)|] = \sum_{j \in \{0,1\}} \mathbb{E}_{\mu}[s_j(f,x,i)] =$$
$$= \sum_{j \in \{0,1\}} \mathbb{E}_{\mu}[f(x)\, s_j(f,x,i)] + \mathbb{E}_{\mu}[(1 - f(x))\, s_j(f,x,i)].$$

We prove (2). Equation (3) follows by the same argument. Fix some true point x' of f with $x'_i = 0$. It has mass $\mu(x')$ and $s_0(f,x',i)$ indicates if a false point y' with $|x' \oplus y'| = 1$ exists. This point has mass $\frac{\mu_i}{1-\mu_i}\,\mu(x')$ since $x'_i = 0$. Moreover, every pair (x,y) of a true point x with $x_i = 0$ and a false point y with $|x \oplus y| = 1$ and $y_i = 1$ contribute the same value $\frac{\mu_i}{1-\mu_i}\,\mu(x) = \mu(y)$ to the sum on the left side of

$$\frac{\mu_i}{1-\mu_i} \sum_{|x \oplus y| = 1} (1 - x_i)\, f(x)\, (1 - f(y))\, \mu(x) = \frac{\mu_i}{1-\mu_i}\, \mathbb{E}_{\mu}[f(x)\, s_0(f,x,i)]$$

and to the sum on the left side of

$$\sum_{|x \oplus y| = 1} y_i\, f(x)\, (1 - f(y))\, \mu(y) = \mathbb{E}_{\mu}[(1 - f(y))\, s_1(f,y,i)].$$

The left sides of these identities are thus equal. This implies that (2) holds and therefore also the lemma. □

We recall Jensen's Inequality.

Proposition 7. *Let $\varphi : R \subseteq \mathbb{R} \to \mathbb{R}$ be convex, $0 \le \lambda_i \le 1$, $\sum_{i=1}^{n} \lambda_i = 1$. Then*

$$\varphi\left(\sum_{i=1}^{n} x_i\, \lambda_i\right) \le \sum_{i=1}^{n} \varphi(x_i)\, \lambda_i. \tag{4}$$

Definition of D. Let F be a k-cnf which defines f and $\mu = (\mu_1, ..., \mu_n)$ a product distribution. We start with defining the randomized algorithm eppz(F) (Extended Paturi & Pudlák & Zane Algorithm [22]). It takes as input F and outputs a point $x \in \{0,1\}^n$. The algorithm chooses uniformly at random some permutation π of the variable indices $[n]$. For $i = 1, ..., n$ the variable $v_{\pi(i)}$ is set to 1 if the single-variable clause $(v_{\pi(i)})$ is in F, set to 0 if $(\overline{v}_{\pi(i)})$ is in F, or — in the last case — the value of $v_{\pi(i)}$ is set to 1 with probability μ_i and to 0 with probability $1 - \mu_i$. At the end x is output. Define for $D = D_F$ the mass of a point x as

$$D_F(x) := \Pr(\text{eppz}(F) \text{ outputs } x).$$

The probability is over the random choices eppz makes. We are only interested in the event that some true point, i.e. satisfying assignment, is output.

Lemma 8 (Extension of [22]). *Let f be a k-cnf function defined by F and x a true point of f. Then*

$$D_F(x) \geq \mu(x) \prod_{i=1}^{n} \left(\frac{1}{\mu_i}\right)^{\frac{s_1(f,x,i)}{k}} \left(\frac{1}{1-\mu_i}\right)^{\frac{s_0(f,x,i)}{k}}. \tag{5}$$

Proof. Denote by $l_j(F,\pi,x,i)$ the indicator the eppz sets the value of variable v_i randomly to j, $j \in \{0,1\}$, given that π was chosen and x output. We prove the following claim first.

$$\mathbb{E}_\pi[l_0(F,\pi,x,i)] \leq (1-x_i)\left(1-\frac{s_0(f,x,i)}{k}\right) = (1-x_i) - \left(\frac{s_0(f,x,i)}{k}\right). \tag{6}$$

$$\mathbb{E}_\pi[l_1(F,\pi,x,i)] \leq x_i\left(1-\frac{s_1(f,x,i)}{k}\right) = x_i - \left(\frac{s_1(f,x,i)}{k}\right). \tag{7}$$

Let x be true point and x' be a false point such that $|x \oplus x'| = 1$ and $x_i \neq x'_i$. Since x is a true point and x' a false point there exists a clause $C = \bigvee_{l\in P} v_l \vee \bigvee_{l\in N} \overline{v}_l$ in F such that $C(x') = 0$ and $C(x) = 1$. Since $|x \oplus x'| = 1$ the only literal in C set to 1 by x is the literal of variable v_i. (Note that $i \in P$ if $j = 1$ and $i \in N$ if $j = 0$.) Algorithm eppz sets the value of v_i deterministically if i occurs after all indices $P \cup N \setminus \{c\}$ in π. This happens with probability $1/|P \cup N| \geq 1/k$. The existence of C is indicated by $s_j(f,x,i)$. Thus $\Pr_\pi(l_j(F,x,\pi,i) = 0) \geq s_j(f,x,i)/k$. This implies $\Pr_\pi(l_j(F,x,\pi,i) = 1) \leq 1 - s_j(f,x,i)/k$ whether $x_i = 0$ or $x_i = 1$. The probability is however 0 in one of the cases which we express by a factor x_i and $1 - x_i$, respectively. This proves the claim.

Let P be the set of all permutations of the variable indices $[n]$ and $x \in \{0,1\}^n$. Using Jensen's Inequality (4) and then (6) yields

$$\begin{aligned}
D_F(x) &= \sum_{\sigma\in P} \Pr(\mathrm{eppz}(F) = x \,|\, \pi = \sigma)\, \Pr(\pi = \sigma) = \\
&= \sum_{\sigma\in P} \prod_{i=1}^{n} \mu_i^{l_1(F,\sigma,x,i)} (1-\mu_i)^{l_0(F,\sigma,x,i)} \frac{1}{n!} \geq^{(4)} \\
&\geq 2^{\mathbb{E}_\pi[\sum_{i=1}^{n} \log(\mu_i)\, l_1(F,\pi,x,i)) + \log(1-\mu_i)\, l_0(F,\pi,x,i)]} = \\
&= \prod_{i=1}^{n} \mu_i^{\mathbb{E}_\pi[l_1(F,\pi,x,i)]} (1-\mu_i)^{\mathbb{E}_\pi[l_0(F,\pi,x,i)]} \geq^{(6),(7)} \\
&\geq \prod_{i=1}^{n} \mu_i^{x_i - \frac{s_1(f,x,i)}{k}} (1-\mu_i)^{(1-x_i) - \frac{s_0(f,x,i)}{k}} = \\
&= \mu(x) \prod_{i=1}^{n} \mu_i^{-\frac{s_1(f,x,i)}{k}} (1-\mu_i)^{-\frac{s_0(f,x,i)}{k}}.
\end{aligned}$$

□

Concentration on True Points. We are now in a position to make our intuition precise. We provide a lower bound for $\mathbb{E}_{D_F}[f]$ and use it to prove an upper bound on $\mathcal{I}^{\mu}(f)$. We prove actually a more general result than Theorem 1 to emphasize a counter intuitive fact. Hard instances of eppz have small total influence. Comparing with results from circuit complexity [4,16] and PAC-learning [16,7] we would rather guess the opposite. In the definition of hard instances we take into account that the success probability p_F of eppz, i.e. $p_F = \mathbb{E}_{x\sim D_F}[f(x)]$, depends on $\mu(f)$. In particular, $p_F \geq \mu(f)$ holds.

Definition 9. *Let p_F be the success probability of* eppz *given a k-cnf F as an input. Let f be the function defined by F. F is called* c-hard *iff $p_F \leq c\,\mu(f)$. A Boolean function g is called (c,k)-hard iff there exists a k-cnf G defining g such that G is* c-hard.

Theorem 10. *Let μ be a product distribution and f be a (c,k)-hard function with $\mu(f) > 0$. Then,*

$$\mathcal{I}^{\mu}(f) \leq u_{\mu}\, k\, \mu(f)\, \log(c).$$

Theorem 1 follows because any k-cnf function is $(1/\mu(f), k)$-hard.

Proof (of Theorem 10). Define

$$S := \mathbb{E}_{\mu}\left[f(x)\left(\sum_{i=1}^{n} \log(1/\mu_i)\, s_1(f,x,i) + \log(1/(1-\mu_i))\, s_0(f,x,i)\right)\right].$$

We have

$$\begin{aligned}
\mathbb{E}_{D_F}[f] &\geq^{(5)} \mathbb{E}_{\mu}\left[f(x)\prod_{i=1}^{n}\left(\frac{1}{\mu_i}\right)^{\frac{s_1(f,x,i)}{k}}\left(\frac{1}{1-\mu_i}\right)^{\frac{s_0(f,x,i)}{k}}\right] = \\
&= \mu(f)\,\mathbb{E}_{\mu}\left[\frac{1}{\mu(f)}\, f(x)\prod_{i=1}^{n}\left(\frac{1}{\mu_i}\right)^{\frac{s_1(f,x,i)}{k}}\left(\frac{1}{1-\mu_i}\right)^{\frac{s_0(f,x,i)}{k}}\right] \geq^{(4)} \\
&\geq \mu(f)\, 2^{S\,(k\,\mu(f))^{-1}}.
\end{aligned}$$

By the prerequisites, there exists a k-cnf F such that

$$\mathbb{E}_{D_F}[f] \leq c\,\mu(f),$$

and therefore $S \leq k\,\mu(f)\,\log(c)$. It it is worthwhile to take a second look at this calculation and note that the application of Jensen's Inequality is responsible for the factor $\mu(f)\,\log(c)$ in our final result. We are left with relating S and $\mathcal{I}^{p}(f)$. By linearity of expectation

$$S = \sum_{i=1}^{n} \log(1/\mu_i)\,\mathbb{E}_{\mu}[f(x)\, s_1(f,x,i)] + \log(1/(1-\mu_i))\,\mathbb{E}_{\mu}[f(x)\, s_0(f,x,i)].$$

By Lemma 6

$$\mathcal{I}^{\mu}(f) = \sum_{i=1}^{n} \frac{1}{\mu_i} \mathbb{E}_{\mu}[f(x)\, s_1(f,x,i)] + \frac{1}{1-\mu_i} \mathbb{E}_{\mu}[f(x)\, s_0(f,x,i)].$$

We want a constant $c = c(\mu)$ s.t. $\mathcal{I}^{\mu}(f) \leq c\, S$. A sufficient condition is given by the inequalities

$$\left(\frac{1}{\mu_i} - c\, \log(1/\mu_i)\right) \mathbb{E}_{\mu}[f(x)\, s_1(f,x,i)] \leq 0$$

and

$$\left(\frac{1}{1-\mu_i} - c\, \log(1/(1-\mu_i))\right) \mathbb{E}_{\mu}[f(x)\, s_0(f,x,i)] \leq 0$$

for all $i \in [n]$. Setting $c = u_{\mu}$ finishes the proof of Theorem 1. □

Remarks

- The combinatorial picture: The quantity $n - \mathcal{I}(f)\,(2\mu_{1/2}(f))^{-1}$ is the average degree of the graph which consists of all true points of f and in which we draw an edge between two vertices iff they have Hamming distance 1 to each other. If the graph emerges from a k-cnf function then it is average degree is at least $n - k \log(1/\mu_{1/2}(f))$.
- Tightness: We define the parity function on k out of n variables as $\mathrm{PAR}_n^k(x) := 1 \oplus x_1 \oplus \ldots \oplus x_k$. Clearly, $\mu_{1/2}(\mathrm{PAR}_n^k) = \frac{1}{2}$ and $\mathcal{I}(\mathrm{PAR}_n^k) = k$. We recall the Edge-Isoperimetric Inequality: $\mathcal{I}(f) \geq 2\, \mu_{1/2}(f)\, \log(1/\mu_{1/2}(f))$ for any Boolean function f. The functions PAR_n^1 and PAR_n^k witness that the Edge-Isoperimetric Inequality and our upper bound are tight for functions with $\mu_{1/2}(f) = \frac{1}{2}$, so called balanced functions. In particular, the gap between the smallest possible and the largest possible total influence of k-cnf functions has to be exactly k for balanced functions and if $\mu = \mu_{1/2}$. It is not clear for what other values of $\mu_{1/2}(f)$ and k this holds. However, $\mathcal{I}(f) = 2\, \mu_{1/2}(f)\, \log(1/\mu_{1/2}(f))$ for all 1-cnf functions since the bounds match in this case.
- Improvement of Boppana's Bound [4]: A consequence of our upper bound is $\mathcal{I}(f) \leq 1.062\, k$. For seeing this note that the upper bound on $\mathcal{I}(f)$ has the form $2\, z\, \log(1/z)$. The function $z\, \log(1/z)$ takes its maximum in the interval $(0, 1]$ at the inverse Euler constant $1/2.718...$ what we can see by differentiating.
- DNFs: By definition, $\mathcal{I}_i^{\mu}(f) = \mathcal{I}_i^{\mu}(1-f)$. This implies an upper of $\mathcal{I}^{\mu}(f) \leq u_{\mu}\, k\, (1-\mu(f))\, \log(1/(1-\mu(f)))$ for any k-dnf function f.

4 Calculating Variable Influences

Variable influences capture the effect of removing a variable. This is a simple consequence of Lemma 6. Assume we have the given Boolean function f_F defined

by a cnf F. Let $F^{(i)}$ be F after removing x_i. We consider $F^{(i)}$ as a cnf over $n-1$ variables. Thus, $f_{F^{(i)}}$ is a Boolean function of the form $\{0,1\}^{n-1} \to \{0,1\}$. Define $\Delta_i F := \mu(f_F) - \mu(f_{F^{(i)}})$. We observe: If F is satisfiable, then $F^{(i)}$ is satisfiable iff $\Delta_i F < \mu(f_F)$.

Lemma 11. *Let $i \in [n]$ and F be a cnf which defines f.*

$$\mathcal{I}_i^{\mu}(f) = \sum_{j\in\{0,1\}} \Pr_{x\sim\mu}(f(x) = 1 \,|\, x_i = j) - \Pr_{x\sim\mu}(f(x) = f(x \oplus e_i) = 1 \,|\, x_i = j).$$

Proof. We recall (1) (Lemma 6). It holds that

$$\mathcal{I}_i^{\mu}(f) = \frac{1}{1-\mu_i}\,\mathbb{E}_{\mu}[f(x)\, s_0(f,x,i)] + \frac{1}{\mu_i}\,\mathbb{E}_{\mu}[f(x)\, s_1(f,x,i)].$$

The claim follows then from

$$\begin{aligned}
&\mathbb{E}_{\mu}[f(x)\, s_j(f,x,i)] = \Pr_{x\sim\mu}(x_i = j, f(x) = 1, f(x \oplus e_i) = 0) = \\
&= \Pr_{x\sim\mu}(x_i = j, f(x) = 1) - \Pr_{x\sim\mu}(x_i = j, f(x) = 1, f(x \oplus e_i) = 1) = \\
&= \Pr_{x\sim\mu}(x_i = j)\left(\Pr_{x\sim\mu}(f(x) = 1 \,|\, x_i = j) - \Pr_{x\sim\mu}(f(x) = f(x \oplus e_i) = 1 \,|\, x_i = j)\right).
\end{aligned}$$

□

We get for the uniform distribution,

$$\mathcal{I}_i(f) = 2(\,\mu_{1/2}(f) - \Pr_{x\sim\mu_{1/2}}(f(x) = f(x \oplus e_i) = 1)).$$

Corollary 12. *Let $i \in [n]$ and F be a cnf which defines f.*

$$\Delta_i F = \frac{\mathcal{I}_i(f)}{2}.$$

Proof. Clearly, $\mu_{1/2}(f_{F^{(i)}}) = \Pr_{x\sim\mu_{1/2}}(f_F(x) = f_F(x \oplus e_i) = 1)$. □

We noted this result because it allows us to compute variable influences in cnfs without designing a new algorithm. If we can compute the involved probabilities, we can compute variable influences and thus the total influence. If $\mu = \mu_{1/2}$, the problem reduces to counting by the previous corollary. An approach of Gomes et al. [8] uses SAT-solvers to approximately count satisfying assignments. Their approach outperforms exact counting algorithms on the tested instances. If the Boolean function is not given by a cnf, we may use the algorithm in [1]. Also note that the theorem yields tractable special cases. An important tractable special case of #SAT are cnfs with bounded tree-width. See for example [6].

Further Remarks

- Complexity: The problem of computing variable influences in cnfs w.r.t. $\mu_{1/2}$ is #P-completeness. This follows directly from the observation that $\mathcal{I}_{n+1}(f_{F'}) = 2\mu_{1/2}(f_{F'}) = \mu_{1/2}(f_F)$ for a cnf F over variables $x_1, ..., x_n$ and $F' = F \wedge x_{n+1}$. For the #P-completeness of computing the total influence of a cnf we set $F' = F \wedge \bigwedge_{i=1}^{n}(x_i \vee x_{n+i}) \wedge (x_i \vee \overline{x}_{n+i})$. If τ is the number of satisfying assignments of F' then $\frac{\tau}{2n}$ is the number of satisfying assignments of F.
- Combining our upper bound with the observation that $F^{(i)}$ is satisfiable iff $\Delta_i F < \mu_{1/2}(f_F)$, we can remove any of at least $n - k \log(1/\mu_{1/2}(f_F))$ variables to get a satisfiability equivalent k-cnf.

5 A Special Case of a Conjecture of Kahn and Kalai

Our proof of Theorem 5 is based on the concept of a canonical decision tree used by Beame [3] to give a combinatorial proof of Håstad's Switching Lemma.

Canonical Decision Tree. Let F be a cnf. Fix an order of the variables. The *canonical decision tree* for F, $T(F)$ is defined inductively as follows: 1. If F is constant 0 or 1 then $T(F)$ consists of a single leaf node labeled by the corresponding constant value 0 or 1. 2. Let C a clause in F and V be the set of variables in C. $T(F)$ starts with a complete binary tree for V. Each leaf l_α in the tree is associated with a point $\alpha \in \{0,1\}^V$ corresponding to the the path from the root to l_α. For each l_α we replace the leaf node l_α by the subtree $T(F_\alpha)$. F_α is F with the variables set according to α.

Let F be a k-cnf defining f. We construct a point $y \in \{0,1\}^I$, $I \subseteq [n]$, s.t.

$$\Pr_{x \sim \mu_{1/2}}(f(x) = 1 \mid x_i = y_i \; \forall i \in I) \geq (1+\delta)\,\mu_{1/2}(f). \tag{8}$$

The point y is described by a path from the root down $T(f)$.

Construction. We go in the canonical decision tree from the root to some inner node or leaf. If we are in situation (1) we stop. In situation (2) we set $I \leftarrow I \cup V$ and extend y by $\alpha^* \in \{0,1\}^V$ s.t. the bias of the function defined by F_{α^*} is maximized. We stop as soon as 8 is satisfied.

Case (1) is clear. So, let C and V be as in Case (2). There are $\alpha_1, ..., \alpha_{2^{|V|}-1} \in \{0,1\}^V$ points satisfying C. One point leads directly to the leaf node 0. For every $i \in [2^{|V|}-1]$ let $f_i : \{0,1\}^n \to \{0,1\}$ be the function resulting from f after fixing the values of the coordinates in V according to α_i. Then

$$\sum_{i=1}^{2^{|V|}-1} \mu_{1/2}(f_i) = \mu_{1/2}(f).$$

By averaging there exists $j \in [2^{|V|} - 1]$ such that

$$\mu_{1/2}(f_j) \geq \frac{\mu_{1/2}(f)}{2^{|V|} - 1}.$$

Thus

$$\Pr_{x \sim \mu_{1/2}} (f(x) = 1 \,|\, x_i = \alpha_i^* \, \forall i \in V) \geq \mu_{1/2}(f) \, \frac{2^{|V|}}{2^{|V|} - 1} \geq \mu(f) \, \frac{2^k}{2^k - 1}.$$

Let V_t be the union of all variable index sets after t steps and let $y \in \{0,1\}^{V_t}$ describe the path gone so far. Then

$$\Pr_{x \sim \mu} (f(x) = 1 \,|\, x_i = y_i \, \forall i \in V_t) \geq \mu(f) \left(\frac{2^k}{2^k - 1} \right)^t.$$

Define t as the smallest integer larger than $\log(1 + \delta) / \log(2^k/(2^k - 1))$. This proves the existence of a point $y \in \{0,1\}^I$ which satisfies (8) and $|I| \leq t\,k = O(k\, 2^k \, \log(1 + \delta))$.

We did not use the monotonicity of f yet. So, let f be monotone increasing. The theorem states the existence of I and $\tilde{y} = (1, ..., 1) \in \{0,1\}^I$ rather than an arbitrary point $y \in \{0,1\}^I$. Since f is monotone increasing

$$\Pr_{x \sim \mu_{1/2}} (f(x) = 1, x_i = 1 \; \forall i \in I) \geq \Pr_{x \sim \mu_{1/2}} (f(x) = 1, x_i = y_i \; \forall i \in I).$$

Together with $\mu_{1/2}(y) = \mu_{1/2}(\tilde{y})$, the theorem follows.

References

1. Bachrach, Y., Markakis, E., Procaccia, A.D., Rosenschein, J.S., Saberi, A.: Approximating power indices. In: Proc. of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 943–950 (2008)
2. Bachrach, Y., Rosenschein, J.S.: Computing the Banzhaf power index in network flow games. In: Proc. of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 335–340 (2007)
3. Beame, P.: A switching lemma primer. Technical Report UW-CSE-95-07-01, Department of Computer Science and Engineering, University of Washington (November 1994)
4. Boppana, R.B.: The average sensitivity of bounded-depth circuits. Information Processing Letters 63(5), 257–261 (1997)
5. Calabro, C., Impagliazzo, R., Kabanets, V., Paturi, R.: The complexity of Unique k-SAT: An isolation lemma for k-CNFs. J. Computer and System Sciences 74(3), 386–393 (2008)
6. Fischer, E., Makowsky, J.A., Ravve, E.V.: Counting truth assignments of formulas of bounded tree-width or clique-width. Discrete Applied Mathematics 156(4), 511–529 (2008)
7. Furst, M.L., Jackson, J.C., Smith, S.W.: Improved learning of AC^0 functions. In: Proc. of the 4th Annual ACM Conference on Computational Learning Theory, pp. 317–325 (1991)

8. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting: A new strategy for obtaining good bounds. In: Proc. of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference (2006)
9. Håstad, J.: A slight sharpening of LMN. J. Computer and System Sciences 63(3), 498–508 (2001)
10. Hirsch, E.A.: A fast deterministic algorithm for formulas that have many satisfying assignments. Logic Journal of the IGPL 6(1), 59–71 (1998)
11. Impagliazzo, R., Paturi, R.: On the complexity of k-SAT. J. Computer and System Sciences 62(2), 367–375 (2001)
12. Kahn, J., Kalai, G.: Thresholds and expectation thresholds. Combinatorics, Probability and Computing 16(3), 495–502 (2007)
13. Kahn, J., Kalai, G., Linial, N.: The influence of variables on boolean functions. In: Proc. of the 29th Annual IEEE Symposium on Foundations of Computer Science, pp. 68–80 (1988)
14. Kalai, G., Safra, S.: Threshold phenomena and influence, with some perspectives from mathematics, computer science, and economics. Discussion Paper Series dp398, Center for Rationality and Interactive Decision Theory, Hebrew University, Jerusalem (August 2005)
15. Klinz, B., Woeginger, G.J.: Faster algorithms for computing power indices in weighted voting games. Mathematical Social Sciences 49(1), 111–116 (2005)
16. Linial, N., Mansour, Y., Nisan, N.: Constant depth circuits, Fourier transform, and learnability. J. ACM 40(3), 607–620 (1993)
17. Matsui, Y., Matsui, T.: NP-completeness for calculating power indices of weighted majority games. Theoretical Computer Science 263(1-2), 305–310 (2001)
18. Nisan, N., Szegedy, M.: On the degree of boolean functions as real polynomials. Computational Complexity 4, 301–313 (1994)
19. O'Donnell, R.: Some topics in analysis of boolean functions. In: Proc. of the 40th Annual ACM Symposium on Theory of Computing, pp. 569–578 (2008)
20. O'Donnell, R., Saks, M.E., Schramm, O., Servedio, R.A.: Every decision tree has an influential variable. In: Proc. of the 46th Annual IEEE Symposium on Foundations of Computer Science, pp. 31–39 (2005)
21. O'Donnell, R., Wimmer, K.: Approximation by DNF: Examples and counterexamples. In: Proc. of the 34th International Colloquium on Automata, Languages and Programming, pp. 195–206 (2007)
22. Paturi, R., Pudlák, P., Zane, F.: Satisfiability coding lemma. Chicago J. Theoretical Computer Science 1999(115) (1999)
23. Prasad, K., Kelly, J.S.: NP-completeness of some problems concerning voting games. International Journal of Game Theory 19(1), 1–9 (1990)
24. Trevisan, L.: A note on approximate counting for k-DNF. In: Proc. of the 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, and 8th International Workshop on Randomization and Computation, pp. 417–426 (2004)

Clause-Learning Algorithms with Many Restarts and Bounded-Width Resolution*

Albert Atserias[1], Johannes Klaus Fichte[2], and Marc Thurley[2]

[1] Universitat Politècnica de Catalunya, Barcelona, Spain
[2] Humboldt Universität zu Berlin, Berlin, Germany

Abstract. We offer a new understanding of some aspects of practical SAT-solvers that are based on DPLL with unit-clause propagation, clause-learning, and restarts. On the theoretical side, we do so by analyzing a concrete algorithm which we claim is faithful to what practical solvers do. In particular, before making any new decision or restart, the solver repeatedly applies the unit-resolution rule until saturation, and leaves no component to the mercy of non-determinism except for some internal randomness. We prove the perhaps surprising fact that, although the solver is not explicitely designed for it, it ends up behaving as width-k resolution after no more than n^{2k+1} conflicts and restarts, where n is the number of variables. In other words, width-k resolution can be thought as n^{2k+1} restarts of the unit-resolution rule with learning. On the experimental side, we give evidence for the claim that this theoretical result describes real world solvers. We do so by running some of the most prominent solvers on some CNF formulas that we designed to have resolution refutations of width k. It turns out that the upper bound of the theoretical result holds for these solvers and that the true performance appears to be not very far from it.

1 Introduction

The discovery of a method to introduce practically feasible clause learning to DPLL-based solvers [15,10] layed the foundation of what is sometimes called "modern" SAT-solving. These methods set the ground for new effective implementations [11] that spawned tremendous gains in the efficiency of SAT solvers with many practical applications. Such great and somewhat unexpected success seemed to contradict the widely assumed intractability of SAT, and at the same time uncovered the need for a formal understanding of the capabilities and limitations underlying these methods.

Several different approaches have been suggested in the literature for developing a rigorous understanding. Among these we find the proof-complexity approach, which captures the power of SAT solvers in terms of propositional proof systems [3,4,9], and the rewriting approach, which provides a useful handle to reason about the properties of the underlying algorithms and their correctness

* Partially supported by project CICYT TIN2007-68005-C04-03.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 114–127, 2009.

[12]. In both approaches, SAT solvers are viewed as algorithms that search for proofs in some underlying proof system for propositional logic. With this view in mind, it was illuminating to understand that the proof system underlying modern solvers is always a subsystem of resolution [3]. In particular, this means that their performance can never beat resolution lower bounds, and at the same time it provides many explicit examples where SAT solvers require exponential time. Complementing this is the observation that an idealized SAT solver that relies on non-determinism to apply the techniques in the best possible way will be able to perform as good as general resolution [4,9]. As the authors in [4] put it, the negative proof complexity results uncover examples of inherent intractability even under perfect choice strategies, while the positive proof complexity results give hope of finding a good choice strategy.

In this work we add a new perspective to this kind of rigorous result. On one hand we try to avoid non-deterministic choices on all components of our abstract solver and still get positive proof complexity results. On the other hand, we test the theoretical results experimentally with some of the available solvers, on benchmarks designed for the purpose. Our main finding is that a concrete family of SAT solvers that do not rely on non-determinism besides mild randomness is at least as powerful as bounded width resolution. The precise proof-complexity result is that under the unit-propagation rule, the totally random branching strategy, and a standard learning scheme considered by true solvers, $8k\ln(8n)n^{2k}$ conflicts and deterministic restarts are enough to detect the unsatisfiability of any CNF formula on n variables having a width-k resolution refutation, with probability at least $1/2$. Note that this bound is *not* asymptotic. The experimental results with actual solvers seem to confirm that our model is faithful enough. We discuss these at the end of this introduction.

The theoretical result by itself has some nice consequences, which we shall sketch briefly. First, it is not very surprising that, although not explicitely designed for that purpose, SAT-solvers are able to solve instances of 2-SAT in very reasonable time. The reason for this is that every unsatisfiable 2-CNF formula has a resolution refutation of width two. More strongly, our result can be interpreted as showing that width-k resolution can be simulated by $O(k\log(n)n^{2k})$ rounds of unit-clause propagation. To our knowledge, such a tight connection between width-k resolution and repeated application of "width-one" methods was unknown before. Another consequence is that SAT solvers are able to solve formulas of bounded branch-width (and hence bounded treewidth) in polynomial time. We elaborate on these later in the paper. Finally, from the partial automatizability results in [5], it follows that SAT solvers are able to solve formulas having polynomial-size tree-like resolution proofs in quasipolynomial time, and formulas having polynomial-size general resolution proofs in subexponential time.

Concerning the techniques, it is perhaps surprising that the proof of our main result does not proceed by showing that the width-k refutation is learned by the algorithm. For all we know the produced proof has much larger width. All we show is that every width-k clause in the refutation is *absorbed* by the algorithm,

which means that it behaves as if it had been learned, even though it might not. In particular, if a literal and its complement are both absorbed, the algorithm correctly declares that the formula is unsatisfiable. This analysis is the main technical contribution of this paper, and deviates significantly from [4] and [9].

Before we close this introduction, a few words on the experimental results are in order. We considered six of the most popular available SAT solvers: BerkMin 5.61 [8], MinSAT 2 [7], Siege ver. 4 [14], zChaff 2001.2.17 (32-Bit version), ZChaff 2007.3.12. (64-Bit) [11] and RSat 2.02 [13]. We ran each of these solvers with specially designed families of unsatisfiable formulas. The formulas come parameterized by two integers q and k and are designed in such a way that the number of variables n is roughly k^2q, the number of clauses is roughly $8k^2q$, and they have width-k resolution refutations. The outcome from the experiment appears to be that the number of decisions (and hence conflicts) that the solvers make on those formulas is bounded by a function of the form n^{c_k}, where c_k is a real number that is in fact smaller but comparable to k.

2 Preliminaries

A *literal* is a propositional variable x or its negation $\bar{x}$. We use the notation x^0 for $\bar{x}$ and x^1 for x. Note that x^a is defined in such a way that the *assignment* $x = a$ satisfies it. For $a \in \{0, 1\}$, we also use $\bar{a}$ for $1 - a$, and for a literal $\ell = x^a$ we use $\bar{\ell}$ for x^{1-a}. A *CNF formula* F is a set of clauses which in turn are sets of literals. The width of a clause is the number of literals in it. For two clauses $A = \{x, \ell_1, \ldots, \ell_r\}$ and $B = \{\bar{x}, \ell'_1, \ldots, \ell'_s\}$ we define the *resolvent of A and B* by $\mathrm{Res}(A, B) = \{\ell_1, \ldots, \ell_r, \ell'_1, \ldots, \ell'_s\}$. We further write $\mathrm{Res}(A, B, x)$ if we want to refer to the variable which we *resolve on*. For a clause C, a variable x, and a truth value $a \in \{0, 1\}$, the *restriction* of C on $x = a$ is the constant $\mathbf{1}$ if the literal x^a belongs to C, and the clause obtained from C by deleting any occurrence of the literal x^{1-a} otherwise. We write $C|_{x=a}$ for the restriction of C on $x = a$. A *partial assignment* is a sequence of assignments $(x_1 = a_1, \ldots, x_r = a_r)$ with all variables distinct. If S is a partial assignment and C is a clause, we let $C|_S$ be the result of applying the restrictions $x_1 = a_1, \ldots, x_r = a_r$ to C. Clearly the order does not matter. We say that S *satisfies* C if it sets at least one of its literals to 1; i.e., if $C|_S = \mathbf{1}$. We say that S *falsifies* C if it sets all its literals to 0; i.e., if $C|_S = \emptyset$. If D is a set of clauses, we let $D|_S$ denote the result of applying the restriction S to each clause in D, and removing the resulting $\mathbf{1}$'s. We call $D|_S$ the *residual* set of clauses.

3 Algorithm and Resolution Width

3.1 Definition of the Algorithm

A *state* is a sequence of assignments $(x_1 = a_1, \ldots, x_r = a_r)$ in which all variables are distinct and some assignments are marked as *decisions*. We use the notation $x_i \stackrel{\mathrm{d}}{=} a_i$ to mean that the assignment $x_i = a_i$ is a *decision assignment*.

In this case x_i is called a *decision variable*. The rest of assignments are called *implied assignments*. We use the letters S and T to denote states. The empty state is the one without any assignments.

The algorithm maintains a current state S and a current database of clauses D. There are four modes of operation DEFAULT, CONFLICT, UNIT, and DECISION. Here is what the algorithm is required to do in each mode:

- DEFAULT. Check if S satisfies every clause in D, in which case stop and output SAT together with the current state S. Otherwise, check if S falsifies some clause in D, in which case move to CONFLICT mode. If not all clauses are satisfied and none of the clauses is falsified, move to UNIT mode. Finally, if control reaches this point, move to DECISION mode.
- CONFLICT. Apply the *learning scheme* to add a new clause to D. Then apply the *restart policy* to decide whether to continue further or to restart in DEFAULT mode with S initialized to the empty state and the current D. In case we continue further, find the most recently added (or conflict-causing) decision $x \stackrel{\text{d}}{=} a$ in S, if such exists. If none is found, stop and output UNSAT. If one is found, replace it by $x = \overline{a}$, delete all later assignments from S, and go back to DEFAULT mode.
- UNIT. For any clause in D for which S gives value to all its literals but one, say x^a, add $x = a$ to the current state and go back to DEFAULT mode.
- DECISION. Apply the *branching strategy* to determine a decision $x \stackrel{\text{d}}{=} a$ to be added to the current state, and go back to DEFAULT mode.

The algorithm is started in DEFAULT mode with the empty state as the current state and the given CNF formula F as the current database.

The well-known DPLL-procedure is the special case of this algorithm in which the learning scheme never adds any new clause, the restart policy does not dictate any restart at all, and the branching strategy chooses the first (or any other) variable that is still unset in the current state. Note that unit-propagation is enforced greedily before every decision is made in accordance to practical implementations. Modern SAT-solvers enhance the performance of the DPLL-procedure by implementing non-trivial learning schemes, restart policies, and branching strategies, as well as a technique known as *backjumping*. This is the mechanism by which the solver in CONFLICT mode determines which conflict-causing decision to backtrack on, based on the clause that the learning scheme adds to the database. We discuss our choice for these components of the algorithm in Section 3.3.

3.2 Runs of the Algorithm

Consider a run of the algorithm started in DEFAULT mode with the empty state and initial database D, until a clause is falsified and thus a conflict occurs. Such a run is called a *round started with* D and we represent it by the sequence of states $S_0, \ldots, S_m$ that the algorithm goes through, where S_0 is the empty state and S_m is the state where the falsified clause is found. Note that for $i \in \{1, \ldots, m\}$, the

state S_i extends S_{i-1} by exactly one assignment of the form $x_i = a_i$ or $x_i \stackrel{\text{d}}{=} a_i$ depending on whether UNIT or DECISION is executed at that iteration.

A *partial round* is an initial segment $S_0, \ldots, S_r$ of a round up to a state where one of the following is true for the residual database $D|_{S_r}$: either $D|_{S_r}$ has no clauses left, or $D|_{S_r}$ contains the empty clause, or $D|_{S_r}$ does not contain any unit clause. If one of the first two cases occurs we say that the partial round is *conclusive*. If a partial round is not conclusive we call it *unconclusive*. We say that the partial round *satisfies* a clause if its last state, interpreted as a partial assignment, satisfies it. We say that it *falsifies* it if its last state, interpreted as a partial assignment, falsifies it. Note that a round may neither satisfy nor falsify a clause.

One important feature of partial rounds is that if they are unconclusive, then the residual database $D|_{S_r}$ does not contain unit clauses and, in particular, it is *closed* under unit propagation. This means that for an unconclusive partial round $S_0, \ldots, S_r$ started with D, if A is a clause in D and S_r falsifies all its literals but one, then S_r must satisfy the remaining literal, and hence A as well. Besides those in D, other clauses may have this property, which is important enough to deserve a definition:

Definition 1. *Let D be a set of clauses and let A be a non-empty clause. We say that D* absorbs *A if for every literal ℓ in A and every unconclusive partial round $S_0, \ldots, S_r$ started with D, if S_r falsifies $A \setminus \{\ell\}$, then it satisfies A.*

We argued already that every clause in D is absorbed by D. We give an example showing that D may absorb other clauses. Let D be the database consisting of the three clauses

$$a \vee \bar{b} \qquad b \vee c \qquad \bar{a} \vee \bar{b} \vee d \vee e.$$

In this example, the clause $a \vee c$ is absorbed by D but does not belong to D. Also the clause $\bar{b} \vee d \vee e$ is not absorbed by D (consider the partial round that starts by $d \stackrel{\text{d}}{=} 0, e \stackrel{\text{d}}{=} 0$) but is a consequence of D (resolve the first and the third clause on a).

The following lemma states three nice monotonicity properties of the concept of clause-absorption, where the first is the one that motivated its definition. We omit the proof due to space restrictions.

Lemma 1. *Let D and E be sets of clauses and let A and B be non-empty clauses. The following hold:*

1. *if A belongs to D, then D absorbs A,*
2. *if $A \subseteq B$ and D absorbs A, then D absorbs B,*
3. *if $D \subseteq E$ and D absorbs A, then E absorbs A.*

The following lemma describes how the resolvent of two absorbed clauses might look if it stays unabsorbed. We say that a partial round $S_0, \ldots, S_r$ *branches* in a set of literals C if all decision variables of S_r are variables of C.

Lemma 2. *Let D be a set of clauses, let A and B be two resolvable clauses that are absorbed by D, and let $C = \mathrm{Res}(A, B)$. If C is non-empty and not absorbed by D, then the following hold:*

1. *there is a literal* x^a *which occurs in both* A *and* B,
2. *there is an unconclusive partial round* R *started with* D *that falsifies* $C \setminus \{x^a\}$,
3. *the partial round* R *branches in* $C \setminus \{x^a\}$ *and leaves* x *unassigned, and*
4. *extending* R *by the decision* $x \stackrel{d}{=} \bar{a}$ *yields a conclusive round.*

Proof. Let $A = \{\ell, \ell_1, \ldots, \ell_p\}$ and $B = \{\bar{\ell}, \ell'_1, \ldots, \ell'_q\}$. As C is non-empty and not absorbed by D, there is a literal x^a in C and an unconclusive partial round $T_0, \ldots, T_s$ started with D which falsifies $C' = C \setminus \{x^a\}$ but does not satisfy C. In particular x is not assigned a in T_s. Also x is not assigned $\bar{a}$ in T_s since otherwise, as A and B are absorbed by D, both ℓ and $\bar{\ell}$ would be satisfied by T_s. This shows that x is unassigned in T_s.

Assume without loss of generality that x^a belongs to A, let $A' = A \setminus \{x^a\}$ and define $B' = B \setminus \{\bar{\ell}\}$. We also have that x^a belongs to B. To see this, observe that otherwise T_s would falsify B', implying that ℓ is falsified by T_s as B is absorbed by D. Then T_s falsifies A' and thus x is set to a in T_s, this time because A is absorbed by D. This contradicts the previous argument that x is unassigned in T_s. Altogether we have that x^a occurs in both A and B.

We still need to prove the existence of an unconclusive partial round $S_0, \ldots, S_r$ as stated in the Lemma. We will construct this round from the given unconclusive one $T_0, \ldots, T_s$. As T_s falsifies C' and x is unassigned in T_s the sole issue we need to resolve is the possibility that T_s might not branch in C'. We will define $S_0, \ldots, S_r$ inductively. It will be convenient to also define an offset j_i for each $i \in \{0, \ldots, r\}$. Recall that S_0 is the empty state by definition. We define $j_0 = 0$. To construct S_{i+1}, let $h > j_i$ be the minimum number in $\{0, \ldots, s\}$ such that the h-th assignment in T_s is of one of the following types, if it exists:

1. a decision $y \stackrel{d}{=} b$ for some variable y from C',
2. an implied assignment $y = b$, and $\{y^b\}$ is a unit clause in $D|_{S_i}$,
3. an implied assignment $y = b$, and y is a variable from C'.

If no such h exists, we stop the construction and let $r = i$. If such an h exists, we define $j_{i+1} = h$ and S_{i+1} from S_i by cases. In the first of the three cases above, let S_{i+1} be obtained from S_i by adding the decision $y \stackrel{d}{=} b$. In the second case, the assignment is due to the existence of the unit clause $\{y^b\}$ in $D|_{T_{h-1}}$. As $\{y^b\}$ is also a unit clause in $D|_{S_i}$, we define S_{i+1} as the extension of S_i by adding this assignment. If the first two cases do not occur, we must be in the third, and we define S_{i+1} from S_i by extending it with $y \stackrel{d}{=} b$.

Clearly, this defines a valid partial round $S_0, \ldots, S_r$ which branches in C'. Further S_r falsifies C' and all the assignments in S_r also appear in T_s except for some additional "decision" marks on some assignments. Therefore the partial round is unconclusive and the variable x is unassigned in S_r. Finally, as the assignments S_r and T_s are the same with respect to the literals in A and B, extending the partial round $S_0, \ldots, S_r$ by a decision $x \stackrel{d}{=} \bar{a}$ yields a conclusive round; otherwise both ℓ and $\bar{\ell}$ would be satisfied since both A and B are absorbed by D. □

The following lemma will allow us to turn the existential statement about a partial round R in Lemma 2 into a universal statement about all partial rounds

R' that make decisions that are already in R. If R and R' are partial rounds, we say that the decisions of R' are subsumed by R if every decision assignment in R' is also an assignment in R. We say that R' is subsumed by R if every assignment made in R' is also an assignment in R.

Lemma 3. *Let D be a set of clauses, let R be an unconclusive partial round started with D, and let R' be a partial round started with D with all its decisions subsumed by R. Then R' is subsumed by R.*

Proof. Let $S_0, \ldots, S_r$ and $T_0, \ldots, T_s$ be the partial rounds R and R', respectively. Assume for contradiction that R' is not subsumed by R. Then there exists a minimal $i \in \{1, \ldots, s\}$ such that all assignments made in T_{i-1} are also in made in R, but the last assignment in T_i is not made in R. Since every decision assignment in R' is also an assignment in R, the last assignment in T_i must be an implied one of the form $x = a$. Thus, there exists a unit clause $\{x^a\}$ in $D|_{T_{i-1}}$. As every assignment in T_{i-1} is also made in R, and as R is unconclusive and does not contain $x = a$ or $x \stackrel{\mathrm{d}}{=} a$, there exists a $j \in \{0, \ldots, r\}$ such that this unit clause is also present in $D|_{S_j}$. Finally, since R is an unconclusive partial round, $D|_{S_r}$ does not contain unit clauses and thus $x = a$ is also an assignment in S_r. Contradiction. □

One consequence of this lemma is that under the hypothesis and the notation of Lemma 2, if x^a and R are the literal and the unconclusive partial round claimed to exist in that lemma, then every partial round started with D that has all its decisions subsumed by R stays unconclusive, and if in addition it ends up falsifying $C \setminus \{x^a\}$, then it yields a conclusive round after extending it with $x \stackrel{\mathrm{d}}{=} \bar{a}$. Indeed, the extension $x \stackrel{\mathrm{d}}{=} \bar{a}$ would force the round to satify both ℓ and $\bar{\ell}$ as both A and B are absorbed by hypothesis. We will need this fact in what follows.

3.3 Restart Policy, Learning Scheme, and Branching Strategy

The only really important issue of the *restart policy* that we want to use is that it should dictate restarts often enough. As a matter of fact, we will state and prove our result for the most aggressive of all restart policies, the one that dictates a restart at every conflict, and the analysis will extend to other restart policies by monotonicity. More precisely, by the monotonicity properties discussed in the previous section, it will follow from our analysis that if we decide to use a policy that allows $c > 1$ conflicts per round before a restart, then the upper bound on the number of required restarts can only decrease (or stay the same). Only the upper bound on the number of conflicts would appear multiplied by a factor of c, even though the truth might be that even those decrease as well. One further consequence of monotonicity is that the validity of our analysis is insensitive to whether the solver implements *backjumping* or not. For the same reason as before, allowing $c > 1$ conflicts per round with their corresponding backjumps can only decrease the number of required restarts in our analysis, and multiply

the number of conflicts by a factor of c. Thus, for the rest of the paper, we fix the restart policy to the one that restarts at every conflict.

Let us discuss now the *learning scheme*. This determines which clause to add to the database in the CONFLICT mode of the algorithm. We will consider the scheme called DECISION in the literature, that obtains the clause by the following method. Let $S_0, \ldots, S_m$ be a conclusive round started with the clause database D that ends up falsifying some clause of D. We anotate each state S_i of the round by a clause A_i by reverse induction on $i \in \{1, \ldots, m\}$:

1. For $i = m$, let A_i be the first clause in D that is falsified by S_i.
2. For $i < m$ for which $x_i \stackrel{\text{d}}{=} a_i$ is a decision, let $A_i = A_{i+1}$.
3. For $i < m$ for which $x_i = a_i$ is implied, let B_i be the first clause in D which contains literal $x_i^{a_i}$ and for which S_{i-1} gives value to all its literals but one, and let $A_i = \text{Res}(A_{i+1}, B_i, x_i)$ if these clauses are resolvable on x_i, and let $A_i = A_{i+1}$ otherwise.

It is quite clear from the construction that each A_i has a resolution proof from the clauses in the database D. In fact, the resolution proof is linear and even trivial in the sense of [4]. The learning scheme called DECISION is the one that adds the clause A_1 to the current database after each conflict. It is not hard to check that every literal in A_1 is the negation of some decision literal in S_m; this will be important later on.

The *branching strategy* determines which literal x^a is branched next in the DECISION mode of the algorithm. We will analyse the totally random branching strategy defined as follows: if the current state of the algorithm is S, we choose a variable x uniformly at random among the variables that appear in the residual database $D|_S$, and a value a in $\{0, 1\}$ also uniformly at random and independently of x. Our analysis actually applies to any other branching strategy that randomly chooses between making a heuristic-based decision or a random decision as above, provided the second case has non-negligible probability of happening. If $p \in (0, 1]$ is the probability of the second case, the bounds in our analysis will appear multiplied by a factor of p^{-k}, where k is the resolution width that we are trying to achieve.

3.4 Resolution Width

We start by analysing the number of rounds it takes until the resolvent of two absorbed clauses is absorbed as a function of its width.

Lemma 4. *Let D be a database of clauses, and let A and B be two resolvable clauses that are absorbed by D and that have a non-empty resolvent $C = \text{Res}(A, B)$. Then, for every integer $t \geq 0$, the probability that C is not absorbed by the database after t restarts is at most $e^{-t/4n^k}$, where n is the total number of variables in D and k is the width of C.*

Proof. Let $D_0, D_1, \ldots, D_t$ be the sequence of databases produced by the algorithm, starting with $D = D_0$. By the monotonicity properties in Lemma 1, if C

is ever absorbed by some D_i it will stay so until D_t. Thus, it will suffice to bound the probability that D_{i+1} does not absorb C conditional on the event that D_i does not absorb C.

Assume D_i does not absorb C. By Lemma 2, there exists a literal x^a in $A \cap B \cap C$ and an unconclusive partial round R started with D_i that falsifies $C \setminus \{x^a\}$, branches in $C \setminus \{x^a\}$, leaves x unassigned, and the extension of R by $x \stackrel{\mathrm{d}}{=} \bar{a}$ yields a conclusive round. Moreover, by Lemma 3 and the discussion after it, any partial round R' that has all its decisions subsumed by R stays subsumed by R and unconclusive, and if it ends up falsifying $C \setminus \{x^a\}$, then its extension by $x \stackrel{\mathrm{d}}{=} \bar{a}$ will also yield a conclusive round. Such a round would yield a conflict that makes the DECISION scheme learn a subclause of C, which implies that C would be absorbed by D_{i+1} by Lemma 1.

First, let us compute a lower bound on the probability that the first $k-1$ choices of the branching strategy falsify $C \setminus \{x^a\}$ and that the k-th choice is $x \stackrel{\mathrm{d}}{=} \bar{a}$. This probability is at least

$$\left[\left(\frac{k-1}{2n}\right)\left(\frac{k-2}{2(n-1)}\right)\cdots\left(\frac{1}{2(n-k+2)}\right)\right]\left(\frac{1}{2(n-k+1)}\right) \geq \frac{1}{4n^k}.$$

Note that a round following these choices may not even be able to do some of the decisions as the corresponding assignments may be implied. However, before the decision $x \stackrel{\mathrm{d}}{=} \bar{a}$, the round will only perform decisions that are subsumed by R and therefore stay subsumed by R by Lemma 3. In particular it will stay unconclusive and x will remain unset. It follows that the probability that the round will start by branching in, and falsifying, $C \setminus \{x^a\}$, and end by deciding $x \stackrel{\mathrm{d}}{=} \bar{a}$ can only increase. This gives a lower bound on the probability that a subclause of C is actually learned, and with it, the probability that C is not absorbed by D_{i+1} is bounded by $1 - \frac{1}{4n^k}$.

By chaining these t conditional probabilities, the probability that C is not absorbed by D_t is bounded by

$$\left(1 - \frac{1}{4n^k}\right)^t \leq e^{-t/4n^k},$$

as was to be proved. □

Finally, we are ready to state and prove the main result of the paper.

Theorem 1. *Let F be a set of clauses on n variables having a resolution refutation of width k and length m. With probability at least $1/2$, the algorithm started with F learns the empty clause after at most $4m\ln(4m)n^k$ conflicts and restarts.*

Proof. The resolution refutation must terminate with an application of the resolution rule of the form $\mathrm{Res}(x, \bar{x})$. We will show that for both $\ell = x$ and $\ell = \bar{x}$, the probability that $\{\ell\}$ is not absorbed by the current database after $4m\ln(4m)n^k$ restarts is at most $1/4$. Thus, both $\{x\}$ and $\{\bar{x}\}$ will be absorbed with probability at least $1/2$. If this is the case, it is straightforward that every round of the

algorithm is conclusive. In particular, the round that does not make any decision is conclusive, and in such a case the empty clause is learned.

Let $C_1, C_2, \ldots, C_r = \{\ell\}$ be the resolution proof of $\{\ell\}$ that is included in the width-k resolution refutation of F. In particular $r \leq m-1$ and every C_i is non-empty and has width at most k. Let $D_0, D_1, \ldots, D_s$ be the sequence of databases produced by the algorithm where $s = rt$ and $t = \lceil 4\ln(4r)n^k \rceil$. For every $i \in \{0, \ldots, r\}$, let E_i be the event that every clause in the initial segment $C_1, \ldots, C_i$ is absorbed by D_{it}, and let $\overline{E}_i$ be its negation. Note that $\Pr[\, E_0 \,] = 1$ vacuously and hence $\Pr[\, \overline{E}_0 \,] = 0$. For $i > 0$, we bound the probability that E_i does not hold conditional on E_{i-1} by cases. Let $p_i = \Pr[\, \overline{E}_i \mid E_{i-1} \,]$ be this probability. If C_i is a clause in F, we have $p_i = 0$ by Lemma 1. If C_i is derived from two previous clauses, we have $p_i \leq e^{-t/4n^k}$ by Lemma 4, which is at most $1/4r$ by the choice of t.

The law of total probability gives

$$\begin{aligned}\Pr\left[\, \overline{E}_i \,\right] &= \Pr\left[\, \overline{E}_i \mid E_{i-1} \,\right] \Pr\left[E_{i-1}\right] + \Pr\left[\, \overline{E}_i \mid \overline{E}_{i-1} \,\right] \Pr\left[\, \overline{E}_{i-1} \,\right] \\ &\leq \Pr\left[\, \overline{E}_i \mid E_{i-1} \,\right] + \Pr\left[\, \overline{E}_{i-1} \,\right].\end{aligned}$$

Adding up over all $i \in \{1, \ldots, r\}$, together with $\Pr\left[\overline{E}_0\right] = 0$, gives

$$\Pr\left[\, \overline{E}_r \,\right] \leq \sum_{i=1}^{r} p_i \leq \frac{r}{4r} = \frac{1}{4}.$$

Since the probability that C_r is not absorbed by D_{rt} is bounded by $\Pr[\, \overline{E}_r \,]$, the proof follows. □

The total number of clauses of width k on n variables is bounded by $2^k \binom{n}{k}$, which is at most $2n^k$ for every n and k. Therefore, if F has n variables and a width-k resolution refutation, we may assume that its length is at most $2n^k$. We obtain the following consequence:

Corollary 1. *Let F be a set of clauses on n variables having a resolution refutation of width k. With probability at least $1/2$, the algorithm started with F learns the empty clause after at most $8k\ln(8n)n^{2k}$ conflicts and restarts.*

An application of Corollary 1 is that, even though it is not explicitely defined for the purpose, the algorithm can be used to decide the satisfiability of CNF formulas of treewidth at most k in time $O(k\log(n)n^{2k+2})$. This follows from the known fact that every unsatisfiable formula of treewidth at most k has a resolution refutation of width at most $k+1$ [1,6,2]. If we are interested in producing a satisfying assignment when it exists, we proceed by self-reducibility: we assign variables one at a time, running the algorithm $\log_2(n)+1$ times at each iteration to detect if the current partial assignment cannot be extended any further, in which case we choose the complementary value for the variable. For this we use the fact that if F has treewidth at most k, then $F|_{x=a}$ also has treewidth at most k. Note that each iteration is correct with probability at least $1 - 1/2n$, which means that all iterations are correct with probability at least $1/2$. The running time of this algorithm is $O(k(\log(n))^2 n^{2k+3})$.

4 Experiments on Tseitin Formulas

In this section we will discuss the experiments performed to illustrate our theoretical results. The class of formulas we tested are Tseitin formulas on *trees of k-grids*. To give a precise definition let the *k-grid* be a graph $G_k = (V_k, E_k)$ with vertex set $V_k = \{v_{i,j} \mid i, j \in [k]\}$ and edges $E_k = \{\{v_{i,j}, v_{i',j'}\} \mid |i-j|-|i'-j'| = 1\}$. Let further $k' = \lfloor k/2 \rfloor$ and define $\{v_{1,1}, \ldots, v_{1,k'}\}$ as the set of *top* vertices and $\{v_{k,1}, \ldots, v_{k,k'}\}$ and $\{v_{k,k-k'}, \ldots, v_{k,k}\}$ that of *left* and *right* bottom vertices of G_k. In a given rooted binary tree $\mathcal{T}$ we associate with each node t a distinct labelled k-grid G_t. Then if t has a child t' the top vertices of $G_{t'}$ are merged with the left bottom vertices of G_t by identifying $v'_{1,i}$ with $v_{k,i}$ for all $i \in [k']$. A second child t'' is treated analogously by now merging the right bottom vertices of G_t with the top vertices of $G_{t''}$.

For any tree of k-grids $G = (V, E)$ as defined above, we construct an unsatisfiable Tseitin CNF-formula F_G. The construction is well-known an can be found e.g. in [16]. Note that the number of variables of F_G is roughly $n = k^2|V|$. Further, the resolution width of F_G is at most k.

Randomized formulas. To average running times of SAT solvers on the above formulas, we introduce some randomization. Let $q \in \mathbb{N}$. A *random* binary tree $\mathcal{T}$ contains a root r and is constructed as follows. Then for every node t assume that we know the number $q' > 0$ of its descendants to be constructed. Choose $q'' \leq q'$ u.a.r. and recursively construct two subtrees, one with q'' nodes, the other one with $q' - q''$ nodes. The process stops if $q' = 0$. For $q, k \in \mathbb{N}$ a random Tseitin formula $F_{q,k}$ is a formula F_G for some tree of k-grids G which in turn has been constructed from a random binary tree on q nodes.

4.1 Results

We conducted experiments using several SAT solvers on a Linux machine with a 3.0 GHz Pentium 4 processor and 1 GB of RAM. The solvers tested include BerkMin 5.61 [8], MinSAT 2 [7], Siege ver. 4 [14], zChaff 2001.2.17 (32-Bit version), zChaff 2007.3.12. (64-Bit) [11] and RSat 2.02 [13]. As running times of the solvers increase quickly with the parameter k, we chose to consider different test sets for the different solvers.

Small values $k = 2, \ldots, 5$. For each k we generated instances $F_{q,k}$ with q varying from 1 to 101 in steps of 10 with 100 instances per step. Note that for $k = 5$ and $q = 100$ a formula $F_{q,k}$ already contains about 4000 variables and 14000 clauses.

Average running times for solving instances $F_{q,k}$ with $k = 5$ and $q = 100$ are as small as 20s for RSat, whereas MiniSat timed out after 10000s. We are however not interested in the actual running times of the solver but we aim at quantifying the difficulty (as a function of k) of solving these formulas.

We therefore chose to consider the average number of decisions with respect to the number n of variables of the formulas $F_{q,k}$. Under the hypothesis that for fixed k the number of decisions d is bounded by a polynomial in n, we determined, for each solver and each n the minimum $c = c(k)$ such that $d \leq n^c$.

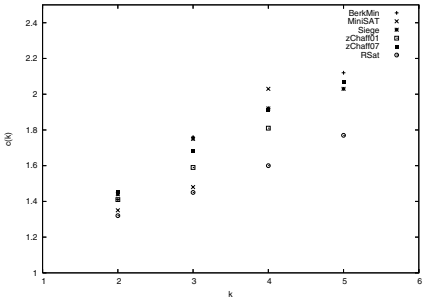


Fig. 1. The value $c = c(k)$

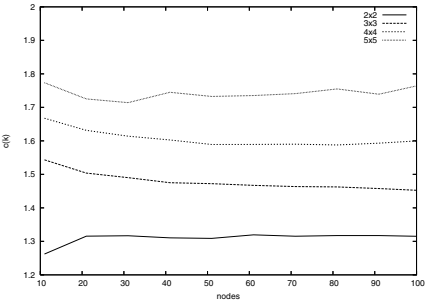


Fig. 2. Stabilization of the Degree

Table 1. The value $c = c(k)$. For the marks * see the discussion in the text.

grid sz.	BerkMin	MiniSAT	Siege	zChaff01	zChaff07	RSat2.0
2×2	1.41	1.35	1.44	1.41	1.45	1.32
3×3	1.76	1.48	1.75	1.59	1.68	1.45
4×4	1.92	2.03	1.92	1.81*	1.91	1.60
5×5	2.12	2.62*	2.03	–*	2.07	1.77
6×6	–	–	2.22	–	2.55*	1.91
7×7	–	–	2.39	–	–	2.04*
8×8	–	–	2.63*	–	–	2.13*

The experimental results show that this c is a function $c = c(k, n)$ of k and n. The dependence of c on n is significant especially for small formulas. However, for fixed k and large n it turns out that the value of $c(k, n)$ is quite stable. For example, on formulas with more than 100 tree nodes we observed that the oscillation of $c(k, n)$ did never exceed 10%. Figure 2 displays the exponents for RSat, which, for comparability, are given in terms of the tree nodes q.

Altogether, it turns out that for fixed k the number of decisions of the solvers is bounded by a polynomial $n^{c(k)}$. Figure 1 illustrates these values of $c(k)$ for the different solvers. The actual values were determined for $q = 100$, which we chose as some solvers turned out to have problems solving much larger instances.

In particular, for $k = 5$ MiniSAT was not able to solve instances with $q \geq 60$ within 10000s therefore the corresponding value has been excluded from Figure 1. Further, although the 2001 zChaff solved many instances for $k = 4, 5$ and arbitrary q within the given time bounds, most of these instances could not be solved due to out-of-memory errors.

Larger values $k = 6, \ldots, 8$. The running times of most solvers quickly exceeded 10000s. Therefore we generated very sparse sets of test instances, mainly to show the tendency of the growth of the running time.

For $k = 6$ we generated a test set with $q = 1, \ldots 101$ in steps of 10 with 10 instances per step. On these formulas BerkMin showed a peculiar behaviour in so far as it was able to solve most instances of up to 6 tree-nodes within 50s although

starting at 7 nodes it was not at all able to solve any instance within 10000s. The 2007 version of zChaff was much more stable, but the average running time exceeded 10000s at 30 tree nodes. The exponent in the table was taken for $q = 51$ where the average running time exceeded even 31000s.

The test set for $k = 7$ was identical to that for $k = 6$. Only Siege and RSat remained for testing. Siege was able to finish the test set. For RSat, the exponent was determined at $q = 81$, since at $q = 91$ out of memory errors occurred. For $k = 8$ we generated a test set of $1, \ldots 51$ tree nodes in steps of 10 and 5 instances per step. The average running time of both Siege and RSat was about 60000s at 51 tree nodes.

Discussion. The test set for smaller k seems to confirm the theoretical results of the previous section. The growth of the decisions of all solvers is polynomial for each fixed k and the exponent of this running time grows at most linearly with k. For Siege and RSat this growth even seems to be mildly sublinear, although exact analysis of this fact would necessitate more detailed tests. Note that the number of decisions is always at least that of the conflicts. Thus the number of conflicts is bounded as predicted by Theorem 1.

However, it is not possible to draw a safe conclusion from these results. Especially by the sparsity of the test set for large k, we cannot take the results to be more than an illustration of the link we assume between true SAT solvers and our theoretical results.

5 Future Work

Our theoretical results establish a correlation between restarts and width, and the experimental results indicate that real-world solvers seem tuned in a way that exploits this correlation. The experiments are however at an early stage and further work will be necessary before drawing definitive conclusions. First, one should try larger test sets and larger values of the parameter k. A second particularly urgent matter is that our experiments do not count restarts directly; they count conflicts, which is only an upper bound on the number of restarts. Related to this is the question of testing the different solvers with different restart policies to compare their behaviour with the theoretical prediction. This is perhaps the most promising open end for applications of our theoretical investigation. Third, an important pressing issue is the lack of a truly general model of randomized formulas of a given width. This is, indeed, a question of theoretical interest by itself.

Acknowledgements

The authors would like to thank Martin Grohe for two reasons. First for giving the idea for the class of formulas used in the experimental part. Most importantly we thank him for the conjecture which became the main result of this paper.

References

1. Alekhnovich, M., Razborov, A.A.: Satisfiability, branch-width and tseitin tautologies. In: FOCS, pp. 593–603. IEEE Computer Society Press, Los Alamitos (2002)
2. Atserias, A., Dalmau, V.: A combinatorial characterization of resolution width. J. Comput. Syst. Sci. 74(3), 323–334 (2008)
3. Beame, P., Kautz, H.A., Sabharwal, A.: Understanding the power of clause learning. In: Gottlob, G., Walsh, T. (eds.) IJCAI, pp. 1194–1201. Morgan Kaufmann, San Francisco (2003)
4. Beame, P., Kautz, H.A., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. J. Artif. Intell. Res (JAIR) 22, 319–351 (2004)
5. Ben-Sasson, E., Wigderson, A.: Short proofs are narrow - resolution made simple. In: STOC, pp. 517–526 (1999)
6. Dalmau, V., Kolaitis, P.G., Vardi, M.Y.: Constraint satisfaction, bounded treewidth, and finite-variable logics. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 310–326. Springer, Heidelberg (2002)
7. Eén, N., Sörensson, N.: An extensible sat-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
8. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat-solver. In: Design, Automation and Test in Europe, DATE 2002 (2002)
9. Hertel, P., Bacchus, F., Pitassi, T., Van Gelder, A.: Clause learning can effectively p-simulate general propositional resolution. In: Fox, D., Gomes, C.P. (eds.) AAAI, pp. 283–290. AAAI Press, Menlo Park (2008)
10. Bayardo Jr., R.J., Schrag, R.C.: Using csp look-back techniques to solve real-world sat instances. In: Proceedings of the Fourtheenth National Conference on Artificial Intelligence (AAAI 1997), pp. 203–208 (1997)
11. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001) (June 2001)
12. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
13. Pipatsrisawat, K., Darwiche, A.: Rsat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA (2007)
14. Ryan, L.: Efficient algorithms for clause-learning sat solvers. Master's thesis, Simon Fraser University (2004)
15. Marques Silva, J.P., Sakallah, K.A.: Grasp - a new search algorithm for satisfiability. In: Proceedings of IEEE/ACM International Conference on Computer-Aided Design, November 1996, pp. 220–227 (1996)
16. Urquhart, A.: Hard examples for resolution. J. ACM 34(1), 209–219 (1987)

An Exponential Lower Bound for Width-Restricted Clause Learning

Jan Johannsen

Institut für Informatik
Ludwig-Maximilians-Universität München
jan.johannsen@ifi.lmu.de

Abstract. It has been observed empirically that clause learning does not significantly improve the performance of a satisfiability solver when restricted to learning short clauses only. This experience is supported by a lower bound theorem: an unsatisfiable set of clauses, claiming the existence of an ordering of n points without a maximum element, can be solved in polynomial time when learning arbitrary clauses, but it is shown to require exponential time when learning only clauses of size at most $n/4$. The lower bound is of the same order of magnitude as a known lower bound for backtracking algorithms without any clause learning. It is shown by proving lower bounds on the proof length in a certain resolution proof system related to clause learning.

1 Introduction

Most contemporary SAT solvers are based on extensions of the basic backtracking procedure known as the DLL-algorithm [6]. One of the most successful of these extensions is clause learning [11], which works roughly as follows: When the backtracking algorithm encounters a conflict, i.e., a clause falsified by the current partial assignment α, then a sub-assignment α' of α that suffices to cause this conflict is computed. This sub-assignment α', the reason for the conflict, can then be stored in form of a new clause C added to the formula, viz. the unique largest clause C falsified by α'. This way the algorithm can later backtrack earlier when again a partial assignment extending α' occurs in another branch of the search tree, since then the added clause C becomes falsified and thus causes a conflict.

When clause learning is implemented, a heuristic is needed to decide which learnable clauses to actually keep in memory, as learning a large number of clauses leads to excessive memory usage, which slows the algorithm down rather than helping it. An obvious simple heuristic is to learn only short clauses, i.e., to set a threshold (possibly depending on the input clauses), and to keep in memory only clauses whose size does not exceed the threshold.

Researchers who have experimented with heuristics for clause learning, e.g. the author himself or Letz [9], have experienced that this simple heuristic is not very helpful, i.e., learning only short clauses does not significantly improve the performance of a DLL algorithm for hard formulas The present work aims at

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 128–140, 2009.

supporting this experience with a rigorous mathematical analysis in the form of a lower bound theorem.

In earlier work [5], we have shown such a lower bound for the well-known pigeonhole principle clauses PHP_n. These formulas require time $2^{\Omega(n \log n)}$ to solve when learning clauses of width up to $n/2$ only, whereas they can be solved in time $2^{O(n)}$ when learning arbitrary clauses. While this example in principle shows the weakness of the heuristic, it is not fully satisfactory, since even with arbitrary learning, the time required is exponential in n, it just takes still more time – about $n!$ – to solve when learning short clauses only.

Here we provide another example using a set of clauses Ord_n based on the ordering principle. These formulas can be solved in polynomial time when learning arbitrary clauses, but require exponential time to solve when learning clauses of size up to $n/4$ only. This lower bound is asymptotically the same as the known exponential lower bound [4] on the time for solving Ord_n by DLL algorithms without clause learning.

The lower bounds on the run-time are shown by proving the same lower bounds on the length of refutations in a certain propositional proof system. The relationship of this proof system to the DLL algorithm with clause learning has been established in several earlier works [5, 7].

2 Preliminaries

A *literal* is a variable x or a negated variable $\bar{x}$, the former are *positive* literals and the latter *negative* literals. A *clause* is a disjunction $C = a_1 \vee \ldots \vee a_k$ of literals a_i, its *width* is $w(C) = k$, the number of literals in it. We identify a clause with the set of literals occurring in it, even though for clarity we still write it as a disjunction. A clause is *negative* if it contains no positive literals. A formula in *conjunctive normal form* (CNF) is a conjunction $F = C_1 \wedge \ldots \wedge C_m$ of clauses, it is usually identified with the set of clauses $\{C_1, \ldots, C_m\}$.

We consider refutation systems for formulas in CNF based on the resolution rule, which are well-known to be strongly related to DLL algorithms. The proof systems under consideration have two inference rules: the *weakening rule*, which allows to conclude a clause D from any clause C with $C \subseteq D$, and the *resolution rule*, which allows to infer the clause $C \vee D$ from the two clauses $C \vee x$ and $D \vee \bar{x}$, provided that the variable x does not occur in either C or D, pictorially:

$$\frac{C \vee x \qquad D \vee \bar{x}}{C \vee D}$$

We say that the variable x is *eliminated* in this inference.

A resolution derivation of a clause C from a CNF-formula F is a directed acyclic graph (dag) with a unique sink, in which every node has in-degree at most 2, and with every node ν labeled with a clause C_ν such that:

1. The sink is labeled with C.
2. If a node ν has one predecessor ν', then C_ν follows from $C_{\nu'}$ by the weakening rule.

3. If a node ν has two predecessors ν_1, ν_2, then C_ν follows from C_{ν_1} and C_{ν_2} by the resolution rule.
4. A source node ν is labeled by a clause C in F.

The *size* of a resolution derivation is the number of nodes in the dag. A *resolution refutation* of F is a resolution derivation of the empty clause from F. We call a derivation *tree-like* if the underlying unlabeled dag is a tree, otherwise we may call it *dag-like* for emphasis.

Note that the weakening rule is redundant in tree-like and dag-like resolution refutations: its uses can be eliminated from a refutation without increasing the size. This may not be the case for the proof system we define below.

A resolution derivation is called *regular* if on every path through the dag, each variable is eliminated at most once. Regularity is not an essential restriction on tree-like resolution since minimal tree-like refutations are always regular [13], but regular dag-like refutations can necessarily be exponentially longer than general ones [1].

Regular tree-like resolution exactly corresponds to the DLL algorithm by the following well-known correspondence: the run of a DLL-algorithm on an unsatisfiable formula F forms a regular, tree-like resolution refutation of F without use of the weakening rule. Since the weakening rule is redundant in tree-like resolution proofs, the converse direction holds as well.

The proof system studied in this work are *resolution trees with lemmas* (RTL), which are defined as follows: An RTL-derivation of C from F is defined like a tree-like resolution derivation of C from F, but here a node with in-degree 2 has a distinguished *left* and *right* predecessor. Then the clause 4 of the definition liberalized to:

4a. A source node ν is labeled by a clause D in F, or by a clause C labelling some node $\nu' \prec \nu$. In the latter case we call C a *lemma*.

Here $\prec$ denotes the post-ordering of the tree, i.e., the order in which the nodes of the tree are visited by a post-order traversal, which at a node ν with two predecessors first recursively traverses the left subtree, i.e., the subtree rooted at the left predecessor of ν, then recursively traverses the right subtree, and then visits ν itself.

An RTL-derivation is an RTL(k)-derivation if every lemma C is of width $w(C) \leq k$. An RTL-derivation of the empty clause from F is an RTL-refutation of F. Note that RTL is equivalent to dag-like resolution and RTL(0) is equivalent to tree-like resolution.

A subsystem WRTI of RTL has been described by Buss et al. [5] which corresponds to a general formulation of the DLL algorithm with clause learning. This system WRTI imposes the regularity restriction on derivations, and does not include the full weakening rule, but incorporates some amount of weakening into a generalized resolution inference rule, the so-called w-resolution rule. It also restricts further the structure of sub-derivations of clauses that can be used as lemmas, which have to be derived by input resolution derivations. W.r.t. the length of proofs, WRTI lies between regular and general dag-like resolution.

The size of a refutation of an unsatisfiable formula F in WRTI has been shown [5] to be polynomially related to the runtime of a schematic algorithm DLL-L-UP on F. This schema DLL-L-UP subsumes all commonly used clause learning strategies, including *first-UIP* [11], *all-UIP*, *decision* [15] and *rel-sat* [2], but is slightly more general than a DLL algorithm with clause learning by being non-greedy in the sense that it can continue branching even after a conflict was reached. In the simulation of clause learning by WRTI, the clauses learned by the algorithm are those clauses used as lemmas in the refutation.

A different system with similar properties was described by Hertel et al. [7], building on earlier work of van Gelder [14], which can likewise be seen as a subsystem of RTL.

It follows that if an unsatisfiable formula F can be solved by a DLL-algorithm with clause learning in time t, then it has an RTL-refutation of size polynomial in t. Moreover, if the algorithm learns only clauses of width k, then the refutation is in RTL(k). In the following we prove lower bounds on the size of refutations in RTL(k), which thus readily translate into lower bounds on the runtime of DLL with width-restricted clause-learning.

A common tool in proof complexity is to consider formulas under a partial assignment, called *restriction* in this context. We shall need a slightly more general notion of restriction in this work.

Let X be a set of variables. A *restriction with renaming* is a (total) function $\rho : X \to X \cup \{0, 1\}$. The function ρ is extended to literals by setting

$$\rho(\bar{x}) := \begin{cases} 1 & \text{if } \rho(x) = 0 \\ 0 & \text{if } \rho(x) = 1 \\ \overline{\rho(x)} & \text{if } \rho(x) \in X \ . \end{cases}$$

For a clause C in variables X, we define

$$C\lceil\rho := \begin{cases} 1 & \text{if } \rho(a) = 1 \text{ for some } a \in C \\ \bigvee_{a \in C,\, \rho(a) \neq 0} \rho(a) & \text{otherwise,} \end{cases}$$

where the empty disjunction is identified with the constant 0. For a CNF-formula F over X, we define

$$F\lceil\rho := \begin{cases} 0 & \text{if } C\lceil\rho = 0 \text{ for some } C \in F \\ \bigwedge_{C \in F,\, C\lceil\rho \neq 1} C\lceil\rho & \text{otherwise,} \end{cases}$$

where the empty conjunction is identified with 1.

Just like ordinary restrictions, the more general renaming restrictions preserve proofs in most propositional proof systems. We state this fact here only for resolution.

Proposition 1. *Let R be a (tree-like) resolution proof of C from F of size s, and ρ a restriction with renaming. Then there is a (tree-like) resolution proof R' of $C\lceil\rho$ from $F\lceil\rho$ of size at most $2s$.*

The proposition is shown by a straightforward induction along the proof R, the proof will not be given here, as we will prove a special case that we actually use below.

In the following we just use the word restriction for restrictions with renaming, since ordinary restrictions do not occur in this work.

3 The Ordering Principle

The ordering principle expresses the fact that every finite total ordering has a maximal element. Its negation is expressed in propositional logic by the following set of clauses Ord_n over the variables $x_{i,j}$ for $1 \leq i, j \leq n$ with $i \neq j$:

$$\begin{array}{lll}
\bar{x}_{i,j} \vee \bar{x}_{j,i} & \text{for } 1 \leq i < j \leq n & (A_{i,j}) \\
x_{i,j} \vee x_{j,i} & \text{for } 1 \leq i < j \leq n & (T_{i,j}) \\
\bar{x}_{i,j} \vee \bar{x}_{j,k} \vee \bar{x}_{k,i} & \text{for } 1 \leq i < j, k \leq n \text{ with } j \neq k & (\Delta_{i,j,k}) \\
\bigvee_{j \in [n] \setminus \{i\}} x_{i,j} & \text{for } 1 \leq i \leq n & (M_i)
\end{array}$$

Let R be the relation on $[n]$ given by an assignment to the variables, so that $i \; R \; j$ holds iff $x_{i,j}$ is set to 1. The clauses $A_{i,j}$ and $T_{i,j}$ state that for every i and j, either $i \; R \; j$ or $j \; R \; i$ holds, but not both. The clause $\Delta_{i,j,k}$ state that there are no cycles of length 3 in R, which modulo the first two families of clauses is equivalent to R being transitive. Thus the first three clause sets state that R is a total ordering. The clauses M_i then state that this ordering has no maximal element, therefore the formula is unsatisfiable.

The formulas Ord_n were introduced by Krishnamurthy [8] as potential hard example formulas for resolution, but short regular resolution refutations for them were constructed by Stålmarck [12].

Proposition 2. *There are dag-like regular resolution refutations of* Ord_n *of size* $O(n^3)$.

Note that the size of the formula Ord_n is $\Theta(n^3)$, so the size of these refutations is linear in the size of the formula. A general simulation of regular resolution by WRTI [5] yields WRTI-refutations of Ord_n of polynomial size. From these, it is straightforward to construct a polynomial length run of a DLL algorithm with clause learning on Ord_n, making the branching and learning decisions suggested by the refutation.

On the other hand, the following lower bound for tree-like resolution refutations of Ord_n was shown by Bonet and Galesi [4]. It implies that a DLL algorithm without clause learning requires exponential time to solve these formulas.

Theorem 3. *Every tree-like resolution refutation of* Ord_n *is of size* $2^{\Omega(n)}$.

More precisely, the lower bound proved by Bonet and Galesi is $\Omega(2^{n/6})$. We shall prove a larger lower bound of $\Omega(2^{n/2})$ below. Our main result is a lower bound on the size of RTL(k)-refutations of the formulas Ord_n.

Theorem 4. *For $k < n/4$, every* RTL(k)*-refutation of* Ord_n *is of size* $2^{\Omega(n)}$.

It follows that a DLL algorithm with learning requires exponential time to solve these formulas, when learning is restricted to clauses of width less than $n/4$.

The idea of the proof is similar to that of the mentioned lower bound for the pigeonhole principle PHP_n [5]: the goal is to show that a long derivation is required to obtain a clause that is short enough to be used as a lemma. To prove this, look at the first sufficiently short clause C, and find a restriction ρ falsifying C. Then the derivation of C, restricted by ρ, is a tree-like resolution refutation of $\mathrm{PHP}_{n'}$ for some $n' < n$, and therefore needs to be large by a known lower bound.

This strategy does not quite work here directly, since from Ord_n short clauses can be derived very quickly. Therefore we single out a class of useful clauses, and show that any refutation can be transformed so that only these useful clauses are used as lemmas, in Section 5.

After that, we again look at the first clause used as a lemma, and find a restriction falsifying it. Thereby we obtain a tree-like refutation of a smaller instance of the ordering principle, which needs to be large by a known lower bound. A class of restrictions that makes this construction possible is defined below.

The argument becomes simpler if the proof is first brought into a normal form that contains only negative clauses; this is done in Section 4. Finally, in Section 6, everything is put together to prove the theorem.

As mentioned, we need to define a class of restrictions that preserve the ordering principle clauses, similar to the matching restrictions that preserve the pigeonhole principle formulas, but in contrast to those we require restrictions with renaming. For a non-empty set $S \subseteq [n]$ and a total ordering $\prec$ on S, we define the *ordering restriction* $\rho(S, \prec)$ by

$$\rho(S,\prec)\ :\quad x_{i,j} \mapsto \begin{cases} 1 & \text{if } i,j \in S \text{ and } i \prec j \\ 0 & \text{if } i,j \in S \text{ and } j \prec i \\ x_{s,j} & \text{if } i \in S \text{ and } j \notin S \\ x_{i,s} & \text{if } i \notin S \text{ and } j \in S \\ x_{i,j} & \text{otherwise,} \end{cases}$$

where $s \in S$ is arbitrary but fixed, e.g. $s := \max S$. We let σ range over ordering restrictions, and for $\sigma = \rho(S, \prec)$ we let $|\sigma| := |S|$.

The main property of ordering restrictions is that they preserve the ordering principle formulas.

Proposition 5. *For every ordering restriction σ with $|\sigma| \geq 1$,*

$$\mathrm{Ord}_n\restriction\sigma \;=\; \mathrm{Ord}_{n-|\sigma|+1}\,.$$

Proof. We shall see that the restriction of every clause from Ord_n by $\sigma = \rho(S, \prec)$ with $|S| \geq 1$ is again one of the clauses from Ord_n, with indices from $[n] \setminus S \cup \{s\}$. Thus after a renaming of variables we obtain the clauses $\mathrm{Ord}_{n-|S|+1}$.

The clauses $T_{i,j}$, $A_{i,j}$ and $\Delta_{i,j,k}$ for $i,j,k \notin S$ remain unaffected by the restriction.

The restriction by σ of the clauses $T_{i,j}$, where $i \in S$ and $j \notin S$ are the clauses $T_{s,j}$, and similarly for $j \in S$ and $i \notin S$. The clauses $T_{i,j}\lceil\sigma$ with $\{i,j\} \subseteq S$ are satisfied. The analogous statements hold for the clauses $A_{i,j}$.

The clauses $\Delta_{i,j,k}\lceil\sigma$ with $i \in S$ and $j,k \notin S$ are $\Delta_{s,j,k}$, and similarly for the other situations where $|\{i,j,k\} \cap S| = 1$.

The clauses $\Delta_{i,j,k}$ with $i,j \in S$ and $k \notin S$ with $j \prec i$ are satisfied by σ, and similarly for the symmetric situations as well as for $\{i,j,k\} \subseteq S$. For $i,j \in S$ with $i \prec j$, the restriction of $\Delta_{i,j,k}$ by σ is $A_{s,k}$, and similarly for the symmetric cases.

Finally, the restriction of M_i for $i \notin S$ is M_i over the smaller domain, for the maximal element i of S under $\prec$ it is M_s, and for other values $i \in S$ it is satisfied. □

4 Negative Calculus

We now define a normal form for RTL-derivations from Ord_n, in form of a *negative calculus* NTL that uses only negative clauses.

For a clause C in the variables of Ord_n, we define a negative clause C^N that is equivalent to C w.r.t. ordering restrictions as follows:

$$\begin{aligned} \bar{x}^N_{i,j} &:= \bar{x}_{i,j} \\ x^N_{i,j} &:= \bar{x}_{j,i} \\ C^N &:= \bigvee_{a \in C} a^N \end{aligned}$$

Observe that $w(C^N) \leq w(C)$ for every clause C, but the translated clause can be strictly smaller, e.g., $(x_{1,2} \vee x_{1,3} \vee \bar{x}_{2,1})^N$ is $\bar{x}_{2,1} \vee \bar{x}_{3,1}$. The negative translation Ord^N_n of the ordering principle is the conjunction of the clauses:

$$\begin{array}{ll} A_{i,j} & \text{for } 1 \leq i < j \leq n, \\ \Delta_{i,j,k} & \text{for } 1 \leq i < j,k \leq n \text{ with } j \neq k, \text{ and} \\ M^N_i & \text{for } 1 \leq i \leq n. \end{array}$$

It is easily seen that the negative translation commutes with ordering restrictions, i.e., for every clause C and ordering restriction σ we have $C^N\lceil\sigma = (C\lceil\sigma)^N$. It follows from Lemma 5 and this fact that ordering restrictions preserve the negative-translated ordering principle:

Corollary 6. *For every ordering restriction σ with $|\sigma| \geq 1$,*

$$\mathrm{Ord}^N_n\lceil\sigma \;=\; \mathrm{Ord}^N_{n-|\sigma|+1}\,.$$

In the negative calculus NTL, the essential positive clauses $T_{i,j}$ in the ordering principle are coded in an inference rule, the *negative inference*:

$$\frac{C \vee \bar{x}_{i,j} \qquad D \vee \bar{x}_{j,i}}{C \vee D}$$

An NTL-derivation is defined exactly as an RTL-derivation, only with the negative inference replacing the resolution inference. An NTL-derivation that does not use any lemmas is called a tree-like negative derivation. Also, an NTL-derivation is an NTL(k)-derivation if every lemma used is of width at most k.

We now provide a translation of RTL-derivations from the ordering principle clauses into the negative calculus that preserves the proof size and the width of lemmas used.

Lemma 7. *If C has an* RTL(k)*-derivation from* Ord_n *of size s, then C^N has an* NTL(k)*-derivation from* Ord_n^N *of size at most $2s$.*

Proof. Let R be an RTL(k)-derivation of C from from Ord_n. We construct an NTL(k)-derivation of C^N of the appropriate size.

For each clause C in Ord_n, the translation C^N is in Ord_n^N, so the claim holds for the axiom leaves. For the lemma leaves, we shall take care in the construction that the clauses C^N for C occurring in R, occur in R^N in the same order, so the lemmas can be used as needed. Also note that since $w(C^N) \leq w(C)$, the lemmas used do not exceed the width bound.

If D is derived by a weakening inference from $C \subseteq D$, and C has a derivation of size $s-1$, then by induction C^N has an NTL(k)-derivation of size at most $2s-2$, and a weakening inference yields $D^N \supseteq C^N$. The size of the obtained derivation is at most $2s-1$, and the ordering of clauses in the derivation is preserved.

Now let $C \vee D$ be derived by a resolution inference from $C \vee x_{i,j}$ and $D \vee \bar{x}_{i,j}$, which are derived by RTL(k)-derivations of size s_1 and s_2, resp., where $s = s_1 + s_2 + 1$. By induction, there are NTL(k)-derivations of $\tilde{C} \vee \bar{x}_{j,i}$ of size at most $2s_1$, and of $\tilde{D} \vee \bar{x}_{i,j}$ of size at most $2s_2$, where $\tilde{C} \subseteq C^N$ and $\tilde{D} \subseteq D^N$. A negative inference then yields $\tilde{C} \vee \tilde{D}$, and by a weakening inference we obtain $C^N \vee D^N$. Note that C^N might contain $\bar{x}_{j,i}$, or similarly for D^N, thus we can not necessarily obtain $C^N \vee D^N$ immediately by a negative inference. The size of the derivation is at most $2s_1 + 2s_2 + 2 = 2s$, and the ordering is preserved. □

The converse direction also holds, we state it for completeness without proof since we shall not need it here:

Proposition 8. *If C has an* NTL(k)*-derivation from* Ord_n^N *of size s, then C also has an* RTL(k)*-derivation from* Ord_n *of size at most $6ns$.*

Negative tree-like derivations are preserved under ordering restrictions. Note that this does not hold for arbitrary restrictions.

Proposition 9. *Let R be a tree-like negative derivation of C from F of size s, and σ an ordering restriction. There is a tree-like negative derivation R' of some subclause $C' \subseteq C\lceil\sigma$ from $F\lceil\sigma$ of size at most s.*

Proof. The proof is by induction of s. If $s = 1$, then R is just the single clause $C \in F$, and hence $C\lceil\sigma$ is in $F\lceil\sigma$, having a derivation of size 1 as well.

If C is derived by weakening from $D \subseteq C$, where D has a derivation of size $s-1$, then by the induction hypothesis there is $D' \subseteq D\lceil\sigma$ having a derivation of size at most $s-1$, from which we obtain $C\lceil\sigma \supseteq D\lceil\sigma \supseteq D'$ by a weakening again.

Now let C be derived from $D_1 = D_1' \vee \bar{x}_{i,j}$ and $D_2 = D_2' \vee \bar{x}_{j,i}$ by a negative inference, with D_i having a derivation of size s_i for $i = 1, 2$ where $s = s_1 + s_2 + 1$. By the induction hypothesis, we have for $i = 1, 2$ a derivation of $D_i'' \subseteq D_i\lceil\sigma$ of size at most s_i. We distinguish three cases.

If $\bar{x}_{i,j}$ does not occur in D_1'', then we obtain $C\lceil\sigma \supseteq D_1'\lceil\sigma \supseteq D_1''$ by weakening, and the resulting derivation is of size at most $s_1 + 1$. The case where $\bar{x}_{j,i}$ does not occur in D_2'' is dual.

Otherwise, we have $D_1'' = \tilde{D}_1 \vee \bar{x}_{i,j}$ and $D_2'' = \tilde{D}_2 \vee \bar{x}_{j,i}$, and we obtain $C' = \tilde{D}_1 \vee \tilde{D}_2 \subseteq D_1'\lceil\sigma \vee D_2'\lceil\sigma = C\lceil\sigma$ by a negative inference, giving a derivation of size at most $s_1 + s_2 + 1 = s$ again. □

In particular, if R is a refutation of F, then R' is a refutation of $F\lceil\sigma$. As usual, we denote R' by $R\lceil\sigma$.

We now prove a lower bound on the size of tree-like negative refutations of the (negative-translated) ordering principle that is slightly larger than the bound obtained from the translation of Theorem 3. Via Lemma 7, it yields the same larger lower bound for tree-like resolution refutations of Ord_n. The proof given here is implicit in the proof of a lower bound for regular resolution refutations of a modification of Ord_n [1].

Lemma 10. *Every tree-like negative refutation of* Ord_n^N *is of size at least* $2^{(n-1)/2}$.

Proof. Let R be a tree-like negative refutation of Ord_n^N. We will define a subtree T of R, and for each node ν in T labeled with the clause C_ν an ordering restriction $\sigma_\nu = \rho(S_\nu, \prec_\nu)$ such that $C_\nu\lceil\sigma_\nu = 0$.

The root of T is the root r of R, and we define $S_r = \emptyset$ and $\prec_r$ as the empty ordering. Since $C_r = 0$, the claim holds.

Now suppose we have defined T up to a node ν with $|\sigma_\nu| \leq n-2$. Since no ordering restriction of size less than n falsifies a clause in Ord_n^N, ν must be an inner node in R.

If ν has a single successor ν', and C_ν is derived by weakening from $C_{\nu'} \subset C_\nu$, then $C_{\nu'}\lceil\sigma_\nu = 0$, so we add ν' to T and set $\sigma_{\nu'} = \sigma_\nu$.

If ν has two successors ν_1 and ν_2, and C_ν is derived by a negative inference

$$\frac{C_{\nu_1} \;=\; C \vee \bar{x}_{i,j} \qquad C_{\nu_2} \;=\; D \vee \bar{x}_{j,i}}{C_\nu \;=\; C \vee D}$$

then we distinguish two cases.

If $i \in S_\nu$ and $j \in S_\nu$, then we add one of the children of ν to T. If $i \prec_\nu j$, then we set $\nu' = \nu_1$, otherwise we set $\nu' = \nu_2$, and we add ν' to T. In either case, by construction we have $C_{\nu'}\lceil\sigma_\nu = 0$, and thus we set $\sigma_{\nu'} = \sigma_\nu$.

If $i \notin S$ or $j \notin S$, then we add both ν_1 and ν_2 to T, and in this case we call ν a *branching node*. We set $S_{\nu_1} = S_{\nu_2} = S_\nu \cup \{i, j\}$. We then choose some extension $\prec_{\nu_1} \supseteq \prec_\nu$ with $i \prec_{\nu_1} j$, and another extension $\prec_{\nu_2} \supseteq \prec_\nu$ with $j \prec_{\nu_2} i$. By construction, we have $C_{\nu_i} \restriction \sigma_{\nu_i} = 0$ and $|S_{\nu_i}| \leq |S_\nu| + 2$ for $i = 1, 2$.

Now every branch in T contains at least $(n-1)/2$ branching nodes, and therefore T and hence R is of size at least $2^{(n-1)/2}$. □

5 Cyclic Clauses

For a negative clause C over the variables of Ord_n, let $G(C)$ be the directed graph with vertex set $[n]$ and edges $\{(i, j)\,;\, \bar{x}_{i,j} \in C\}$. A negative clause is *cyclic*, if $G(C)$ contains a (directed) cycle, and acyclic otherwise. It is easily seen that cyclic clauses have short tree-like negative refutations.

Lemma 11. *Any cyclic clause over the variables of* Ord_n *of width* k *has a tree-like negative refutation of size at most* $2\min(k, n)$.

Proof. If $G(C)$ is cyclic, it contains a cycle $i_1, i_2, \ldots i_\ell, i_1$ with $\ell \leq \min(k, n)$. We first show that for every such cycle, the clause

$$\bar{x}_{i_1,i_2} \vee \ldots \vee \bar{x}_{i_{\ell-1},i_\ell} \vee \bar{x}_{i_\ell,i_1}$$

has a negative derivation of length at most $2\ell - 1$. From this clause, the clause C is derived by one weakening inference, hence it has a derivation of length $2\ell \leq 2\min(k, n)$.

We prove the claim by induction on ℓ. For $\ell \leq 3$, this clause is either A_{i_1,i_2} or Δ_{i_q,i_2,i_3}, and hence already in Ord_n^N. Assume the claim holds for ℓ, then by a negative inference we obtain:

$$\frac{\bar{x}_{i_1,i_2} \vee \ldots \vee \bar{x}_{i_{\ell-1},i_\ell} \vee \bar{x}_{i_\ell,i_1} \qquad \bar{x}_{i_1,i_\ell} \vee \bar{x}_{i_\ell,i_{\ell+1}} \vee \bar{x}_{i_{\ell+1},i_1}}{\bar{x}_{i_1,i_2} \vee \ldots \vee \bar{x}_{i_\ell,i_{\ell+1}} \vee \bar{x}_{i_{\ell+1},i_1}}$$

and the length of the resulting derivation is $2\ell - 1 + 2 = 2(\ell + 1) - 1$, which shows the claim. □

It follows that cyclic clauses are useless as lemmas for refuting Ord_n^N.

Lemma 12. *Let* R *be an* $\mathrm{NTL}(k)$*-refutation of* Ord_n^N *of size* s*. Then there is an* $\mathrm{NTL}(k)$*-refutation* R' *of* Ord_n^N *such that every lemma used in* R' *is acyclic, and* $|R'| \leq 2n \cdot s$.

Proof. Replace each cyclic lemma used by its derivation of size at most $2n$, which exists by Lemma 11. □

The final ingredient for our proof is the following lemma showing that a short acyclic clause can always be falsified by a small ordering restriction.

Lemma 13. *If* C *is an acyclic negative clause of width* $w(C) \leq k$*, then there is an ordering restriction* σ *of size* $|\sigma| \leq 2k$ *such that* $C \restriction \sigma = 0$.

Proof. Let S be the set of those $i \leq n$ that are mentioned in C, i.e., such that $\bar{x}_{i,j}$ or $\bar{x}_{j,i}$ occurs in C for some j. Clearly $|S| \leq 2k$. Consider the subgraph G of $G(C)$ induced by S, which only differs from $G(C)$ by omitting isolated vertices. Since C is acyclic, so is G. Let $\prec$ be any topological ordering of G, i.e., a total ordering of S such that $u \prec v$ for every edge (u, v) in G. Then for $\sigma := \rho(S, \prec)$ we have $C\lceil\sigma = 0$ by construction, and $|\sigma| \leq 2k$ as required. □

6 Proof of the Lower Bound

We are now ready to plug all ingredients together to prove our lower bound result, Theorem 4.

Proof. Let $k < n/4$, and let R be an RTL(k)-refutation of Ord_n of size s. By Lemma 7, there is an NTL(k)-refutation R^N of Ord_n^N of size $|R^N| \leq 2s$. Lemma 12 then yields an NTL(k)-refutation R' of Ord_n^N with only acyclic lemmas, of size $|R'| \leq 4ns$.

Let C be the first clause in R' that is used as a lemma. Then the subtree R'_C of R' rooted at C is a tree-like negative derivation of C from Ord_n^N, of size $|R'_C| \leq 4ns$. Since C is acyclic, from Lemma 13 we obtain an ordering restriction σ of size $|\sigma| \leq 2k < n/2$ such that $C\lceil\sigma = 0$, and Proposition 9 yields a tree-like negative refutation $\tilde{R} := R'_C\lceil\sigma$ of $\mathrm{Ord}_{n-|\sigma|+1}^N$ of size at most $8ns$. By Lemma 10, $\tilde{R}$ is of size at least

$$|\tilde{R}| \geq 2^{(n-|\sigma|)/2} \geq 2^{(n-2k)/2} \geq 2^{n/4},$$

therefore we obtain $8ns \geq 2^{n/4}$, and thus

$$s \geq 2^{n/4}/8n = 2^{n/4-\log n-3} = 2^{\Omega(n)}$$

which proves the claim. □

7 Implication Graph Formulas

In contrast to our result above, we now give an example where even the use of very small lemmas gives an exponential speed-up over tree-like resolution. We show that the implication graph formulas for every graph on n vertices have RTL(2)-refutations of linear size, whereas it is known that for some graphs they require exponential size tree-like resolution refutations [3].

Let a *pointed graph* be a directed acyclic graph with a unique sink t, where every vertex that is not a source has in-degree 2. The implication graph formula $Imp(G)$ for such a pointed graph G consists of the source clause $x_s \vee y_s$ for every source s, the sink clauses $\bar{x}_t$ and $\bar{y}_t$, and the four implication clauses

$$\bar{x}_u \vee \bar{x}_v \vee x_w \vee y_w$$
$$\bar{x}_u \vee \bar{y}_v \vee x_w \vee y_w$$
$$\bar{y}_u \vee \bar{x}_v \vee x_w \vee y_w$$
$$\bar{y}_u \vee \bar{y}_v \vee x_w \vee y_w$$

for an inner vertex w with predecessors u and v.

Ben-Sasson et al. [3] show a lower bound for tree-like resolution refutations of the implication graph formulas for certain graphs:

Theorem 14. *There are pointed graphs G_n with n vertices such that tree-like resolution refutations of $Imp(G_n)$ require size $2^{\Omega(n/\log n)}$.*

On the other hand, we have:

Theorem 15. *For every graph G with n vertices, there are* RTL(2)*-refutations of $Imp(G)$ of size $O(n)$.*

Proof. For every vertex w with predecessors u and v, there is a tree-like derivation of $x_w \vee y_w$ from the lemmas $x_u \vee y_u$ and $x_v \vee y_v$ as follows:

First resolve $x_v \vee y_v$ with the first two implication clauses, giving $\bar{x}_u \vee x_w \vee y_w$. Also, resolve $x_v \vee y_v$ with the last two implication clauses to give $\bar{y}_u \vee x_w \vee y_w$. These two are resolved with $x_u \vee y_u$ to obtain $x_w \vee y_w$.

Now these derivations can be plugged together to yield an RTL(2)-derivation of $x_t \vee y_t$ from all the source clauses. Resolving this with the sink clauses gives the desired refutation. □

8 Conclusion

We have provided an example of a class of formulas which can be solved quickly by DLL algorithms with clause learning, but require exponential time when learning is restricted to short clauses. This rigorous lower bound result supports the experience made in practice that restricting to short clauses is not a useful heuristic for deciding which clauses to learn. The hard examples used are the formulas Ord_n based on the ordering principle, which frequently occur as hard examples in proof complexity.

It would be nice to have another example showing this behavior that has only short input clauses, but it seems likely that the technique of this paper can be extended to provide such an example, based on a 3-CNF extension of the formulas Ord_n or a restriction of Ord_n to the edges of an expander graph as used by Segerlind et al. [10]. This is being investigated in ongoing work.

A major problem is to extend the lower bounds to systems with lemmas of arbitrary length, and thus to separate the systems corresponding to DLL with clause learning [5, 7] – and thus the algorithm itself – from general dag-like resolution. For this problem, the techniques used here and in the earlier lower bound for the pigeonhole principle [5] are insufficient, since they rely heavily on the proofs being non-regular. But without the regularity restriction, the systems with arbitrary lemmas are equivalent to general resolution.

Acknowledgments. I thank Jan Hoffmann for helpful discussions about the results in this paper, and two reviewers for some useful suggestions.

References

[1] Alekhnovich, M., Johannsen, J., Pitassi, T., Urquhart, A.: An exponential separation between regular and general resolution. Theory of Computing 3, 81–102 (2007)
[2] Bayardo Jr., R.J., Schrag, R.C.: Using CSP look-back techniques to solver real-world SAT instances. In: Proc. 14th Natl. Conference on Artificial Intelligence, pp. 203–208 (1997)
[3] Ben-Sasson, E., Impagliazzo, R., Wigderson, A.: Near-optimal separation of general and tree-like resolution. Combinatorica 24(4), 585–604 (2004)
[4] Bonet, M.L., Galesi, N.: Optimality of size-width tradeoffs for resolution. Computational Complexity 10(4), 261–276 (2001)
[5] Buss, S.R., Hoffmann, J., Johannsen, J.: Resolution trees with lemmas: Resolution refinements that characterize DLL algorithms with clause learning. Logical Methods in Computer Science 4(4) (2008)
[6] Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Communications of the ACM 5(7), 394–397 (1962)
[7] Hertel, P., Bacchus, F., Pitassi, T., van Gelder, A.: Clause learning can effectively p-simulate general propositional resolution. In: Fox, D., Gomes, C.P. (eds.) Proceedings of the 23rd AAAI Conference on Artificial Intelligence, AAAI 2008, pp. 283–290. AAAI Press, Menlo Park (2008)
[8] Krishnamurthy, B.: Short proofs for tricky formulas. Acta Informatica 22, 253–274 (1985)
[9] Letz, R.: Personal communication
[10] Segerlind, N., Buss, S.R., Impagliazzo, R.: A switching lemma for small restrictions and lower bounds for k-DNF resolution. SIAM Journal on Computing 33(5), 1171–1200 (2004)
[11] Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD), pp. 220–227 (1996)
[12] Stålmarck, G.: Short resolution proofs for a sequence of tricky formulas. Acta Informatica 33, 277–280 (1996)
[13] Tseitin, G.: On the complexity of derivation in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logic, Part 2, pp. 115–125 (1968)
[14] van Gelder, A.: Pool resolution and its relation to regular resolution and DPLL with clause learning. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 580–594. Springer, Heidelberg (2005)
[15] Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD), pp. 279–285 (2001)

Improved Conflict-Clause Minimization Leads to Improved Propositional Proof Traces

Allen Van Gelder

Univ. of California, Santa Cruz, CA 95064
http://www.cse.ucsc.edu/∼avg

Abstract. Recent empirical results show that recursive, or expensive, conflict-clause minimization is quite beneficial on industrial-style propositional satisfiability problems. The details of this procedure appear to be unpublished to date, but may be found in the open-source code of MiniSat 2.0, for example. Biere reports that proof traces are made more complicated when conflict-clause minimization is used because some clauses need to be resolved upon multiple times during the minimization procedure as found in MiniSat 2.0. Biere proposes a proof-trace format in which the set of clause numbers needed for a certain derivation is given, but their order is not specified. This paper presents a new procedure for conflict-clause minimization that is slightly more efficient and, more importantly, discovers a correct order so that each clause used for the derivation is resolved upon only once. This permits the proof trace to specify the order in which to use the clauses, greatly reducing the burden on software that processes the proof trace. The method is validated on the unsatisfiable formulas used for industrial benchmarks in the verified-unsatisfiable track of the SAT 2007 competition.

1 Introduction

Sinz and Biere [6] and later Biere [1] describe and discuss a system of proof traces and checking for Sat solvers based on conflict-driven clause learning, such as `zchaff`, `minisat`, `picosat`, and many others. In many respects, their proposal is simply the union of two earlier ground-breaking proposals: Goldberg and Novikov proposed to output the literals of each derived conflict clause [4], while Zhang and Malik proposed to output the sequence of clause numbers whose linear resolution would create each derived conflict clause [9]. The original motivation for outputting proofs was to provide certificates of correctness that could be checked offline. Biere argues that proofs have other uses in several applications [1]. In these contexts, proofs are viewed as explanations, the main goal is to extract useful information, rather than check correctness. Therefore, the format should be compact and easy to use. Biere argues that the proof trace should contain *both* the conflict clause and the clause numbers needed to derive it.

However, the Sinz and Biere proof-trace format differs in one important respect from other proposals. It produces the *unordered set* of clause numbers that

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 141–146, 2009.

are sufficient to derive the conflict clause. One reason given is that when conflict-clause minimization is employed, the solver knows which clauses are *necessary* for the derivation, but some of the clauses might be used multiple times and figuring out a resolution order would entail extra work. Other inference methods that might appear in the future also might not be amenable to linear derivations.

This short paper shows that it is feasible to produce an ordered sequence, even when "recursive" conflict-clause minimization is employed. Although our procedure is specific for conflict graphs, the idea may well extend to other settings, where more clauses are available. The procedure given in this paper has been "dropped into" `MiniSat 2.0`, replacing the "recursive" conflict-clause minimization in the distribution (called "expensive" in the `MiniSat` code), and has sped up the program slightly. But this minor speed-up is really a by-product. The main motivation is that our procedure is able to discover a correct sequence for deriving a minimized conflict clause by a linear resolution in which no variable is resolved upon more than once, and consequently no clause is used more than once. Such proofs were dubbed *trivial resolutions* by Beame *et al.*. who showed that these proofs are of minimum length among those using only clauses in the conflict graph [2].

After the original and new methods are described, experimental results are presented based on the industrial benchmarks used for the SAT–2007 verification track. See `http://www.cse.ucsc.edu/~avg/ProofChecker/` for details.

2 Conflict Clauses and Conflict Graphs

Leading SAT solvers use a *conflict graph* data structure to infer *conflict clauses*. Readers unfamiliar with conflict graphs and their relationship to conflict clauses should consult citations in this paragraph. Figure 1 illustrates a conflict graph. Our notation varies from some others [10,2] to reflect the data structures used by `zchaff`, `MiniSat`, `picosat`, etc. Arrows indicate the *reference* direction in the data structures, and "$\perp$" is associated with a clause that became *empty* during unit-clause propagation, as in the original presentation [5].

An *antecedent clause* determines the edges leaving the vertex, and *vice versa*. In Figure 1, the antecedent clause of "$\perp$" is $[\overline{e}, \overline{f}, \overline{j}]$, and the antecedent clause of "e" is $[e, \overline{g}, \overline{k}]$.[1] For working through examples, we assume that literals of antecedent clauses are stored in alphabetical order.

In Figure 1, the 1-UIP cut has the associated 1-UIP conflict clause:

$$D_0 = [\overline{p}, \overline{j}, \overline{k}, \overline{i}, \overline{m}, \overline{r}, \overline{\ell}, \overline{q}] \; . \tag{1}$$

The literals of D_0 are listed in the order that `MiniSat` stores them. Notice that those within a decision level are not in any particular order among themselves. This clause and Figure 1 are used for running examples in this paper.

[1] It is easy to see that the set of antecedent clauses for a particular conflict graph is *renamable Horn*, so there is no loss of generality in assuming all vertices correspond to positive literals.

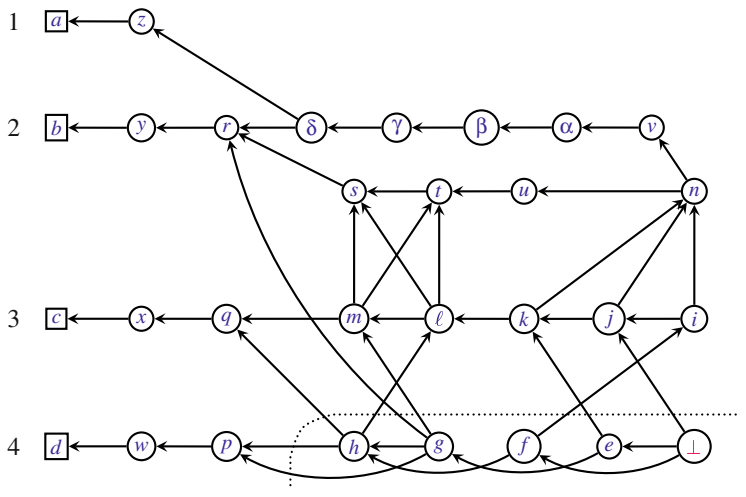


Fig. 1. Conflict graph with 1-UIP cut shown as dotted line. The corresponding conflict clause is $D_0 = [\overline{p}, \overline{j}, \overline{k}, \overline{i}, \overline{m}, \overline{r}, \overline{\ell}, \overline{q}]$. Decision levels and decision literals are on the left.

3 Conflict Clause Minimization in MiniSat 2.0

We now explain the clause minimization procedure in `MiniSat` (and other solvers) using conflict clause D_0 in (1) and Figure 1 as an example. After determining D_0, `MiniSat` 2.0 tests each literal L in D_0, except p, to see if it can be resolved away without (ultimately) adding any new literals to the resulting clause. Only antecedent clauses are considered for such resolutions.[2] The procedure does not actually perform any resolutions. Instead, for each candidate literal L in D_0, a depth-first search rooted at L checks whether all paths leaving L encounter some other vertex in D_0.[3] If this condition holds, L can be removed. The search is aborted as soon as it is determined that some path exists that terminates without meeting D_0. The final conflict clause is called D, and is a subset of D_0.

In the following, the notation "(in)" means the search backtracks from this vertex without exploring further, because this vertex is part of D_0. First, p is bypassed because it is never removable. Next, j is checked, generating the search sequence j, k (in), n, u, t, s, r (in), v, α, β, γ, δ, r (in), z (fail). It is unnecessary to trace the rest of the path from z to a because D_0 has no literals on this decision level, so this must be a failure path. Therefore, j must be kept in D. When the search aborts, all other information found during the search is discarded.

Next, k is checked, generating the search sequence k, ℓ (in), n, ... (the rest is the same as the j search). Then comes i, generating i, j (in), n, ... (the rest is the same as the j search). This repetition shows an inefficiency in the code, but it is not easy to overcome because the depth-first search is coded in

[2] In `MiniSat` code, the vector `reason[]` stores what we call the antecedent clause.

[3] Actually, the negation of the vertex is in the clause, but there is no confusion in the simpler wording.

the old-fashioned manner, without recursion, where the programmer manages the vertex stack. This style has no convenient access to *post-order time* for the vertices (also known as *finishing time*).

The minimization procedure continues with m. The sequence is m, q (in), s, r (in), t, s (removable). Thus m can be removed from D_0. Notice that upon the second encounter with s it was remembered that if s is temporarily added to the clause, it can be removed. The same is remembered about t at this point. As long as the overall search is successful, it remembers that all the vertices visited are removable. It is only when the search ultimately fails that the procedure does not know which vertices are removable and discards all new information.

Next, the search from r proceeds to y and b, and fails, so r must be kept in D. Next, ℓ is checked, generating the sequence ℓ, m (in), s (removable), t (removable). So ℓ is also removable. Finally, q is checked and must be kept in D.

In summary, the procedure found first that m can be removed, then that ℓ can be removed. Unfortunately, a resolution derivation that shortens D_0 by first removing m, then removing ℓ, is unnecessarily long and does not fit the pattern of the trivial resolution. Using the orders found in the searches would lead to the following sequence of resolvents (with abuse of notation): $D_1 = D_0 - m + s + t$, $D_2 = D_1 - s$, $D_3 = D_2 - t + s$, $D_4 = D_3 - s$. Finally, D_4 is D_0 with m removed and nothing added.

But now it gets worse. To remove ℓ, it is necessary to re-introduce m and remove it all over again: $D_5 = D_4 - \ell + m + s + t$, $D_6 = D_5 - m$, $D_7 = D_6 - s$, $D_8 = D_7 - t + s$, $D_9 = D_8 - s$. The final minimized clause is $D = D_9$. Possibly, this example can be extended to construct an exponential worst case [7].

This example explains why Sinz and Biere advocated that the trace should simply specify that D was *somehow* derived from the antecedents of $\bot$, e, f, g, h, m, s, t, and ℓ, without specifying a sequence.

4 New Minimization Procedure

Our new procedure for minimization uses the modern recursive version of depth-first search (DFS) that provides access to the post-order times of vertices (also called finishing time) [3]. The DFS can be visualized as someone moving around the graph and able to do tasks when they arrive at a vertex either for the first time or upon backtracking. Initially vertices are marked as *in* or *out* of D_0. Upon reaching a vertex L at post-order time, enough information has been gathered to categorize it as one of the following: *keep*: L remains in D; *removable*: L is not in D, but is in D_0 or some intermediate resolvent; *poison*: L must not enter any intermediate (or final) resolvent. A decision literal is *keep* if it is in D_0, otherwise *poison*.

The post-order rules for non-decision literals are straightforward: (1) If L is in D_0 and some successor is poison, *keep*; (2) if L is not in D_0 and some successor is poison, *poison*; (3) if all successors of L are keep or removable, L is *removable*.

The crucial idea is this: at the post-order time for L, if L is found to be *removable*, then it is pushed on a *stack*. (Some removables may be unneeded because they cannot be reached by a path of removables; they can be removed

easily by post-processing.) Correctness easily follows using the fact that the top-to-bottom order of the needed removables is a topological order, as required for a trivial resolution to reduce D_0 to D [7].

A DFS is rooted at each L in D_0, but now information is stored for both failing and successful subsearches, so nothing is discarded and repeated.

Let us trace this procedure on the same example. The notation "↓" means the vertex is backtracked to. As before, p is bypassed. A DFS is rooted at j, generating the search sequence j, k, ℓ, m, q, x, c (poison), ↓ x (poison), ↓ q (keep), ↓ m, s, r, y, b (poison), ↓ y (poison), ↓ r (keep), ↓ s (removable, push(s)), ↓ m, t (removable, push(t)), ↓ m (removable, push(m)), ↓ ℓ (removable, push(ℓ)), ↓ k, n, u (removable, push(u)), ↓ n, v, α, β, γ, δ, z (poison), ↓ δ, ↓ γ, ↓ β, ↓ α, ↓ v, ↓ n (all six poison), ↓ k (keep), ↓ j (keep).

A second DFS is rooted at i, but both its successors have been visited, so i is immediately categorized as *keep*.

The top-to-bottom order of the *removable* stack is: u, ℓ, m, t, s. Since D_0 has neither u nor $\overline{u}$, resolution with the antecedent of u is not needed. The trivial resolution to reduce D_0 to D uses the antecedents of ℓ, m, t, s, in that order.

5 Experimental Results

We ran several configurations of `MiniSat` 2.0 on 17 industrial benchmarks from the verified-unsatisfiable track of the SAT 2007 competition. At the URL in Section 1, see `minisat-comparison.pdf` and `cert-poster-sat07.pdf` for additional data and benchmark information; space constraints prevent including them here. CPU times are based on Intel XEON 2.00GHz, 4 GB memory.

The modified `MiniSat` consists of "dropping in" the new recursive conflict-clause minimization procedure presented in this paper (see `MiniSat2ccmin.tar`). This change did not affect the computational results, not even the order of the literals within any clauses, as was verified by observing that all printed counter values (in the 100's per run, including lengths and numbers of conflict clauses at each restart point) were identical for both versions.

Table 1 shows that, on 16 benchmarks (the 17-th was solved by unit propagation), the modification saved time 9 times, and lost time 7 times. However, profiling the `analyze()` percent, the modification lowered this percent 9 times and raised it 2 times. Most changes were only 1 percent of the total time, but the larger changes all favored the modified version and were 2, 3, and 13 percent of the total time. This is the most specific data we could get, because the conflict-clause minimization code is in-line in the original `analyze()`, which also computes the conflict clause, the backtrack level, and other related data. As expected, since no changes were made except in `analyze()` and its subroutines, the larger differences in CPU time are almost entirely attributable to the differences in `analyze()` percent.

This data provides evidence that our improved procedure for "recursive" conflict-clause minimization achieves its primary purpose, which is to enable a proof trace to show a correct order of resolutions to achieve a trivial resolution derivation of the minimized conflict clause, in the sense of Beame *et al.* [2]. Moreover,

Table 1. `MiniSat` 2.0 original and modified CPU times and fractions

benchmark name	CPU orig	CPU mod	CPU mod–orig	percentage mod–orig	`analyze` pct orig	`analyze` pct mod
eq.atree.braun.7.unsat	3.39	3.41	0.02	0.59	28	27
eq.atree.braun.8.unsat	38.34	37.77	-0.57	-1.49	22	22
eq.atree.braun.9.unsat	174.71	180.87	6.16	3.46	16	16
AProVE07-21	1369.26	1362.23	-7.03	-0.50	11	11
AProVE07-02	4204.60	4227.81	23.21	0.55	21	20
AProVE07-22	397.68	395.76	-1.92	-0.47	17	16
AProVE07-20	753.64	764.27	10.63	1.40	13	14
AProVE07-15	609.09	611.34	2.25	0.37	21	21
IBM_FV_2004··30··.k15	1394.55	1357.30	-37.25	-2.70	15	13
itox_vc965	0.25	0.25	0	0	0	0
dated-5-11-u	726.91	738.13	11.22	1.53	8	9
dated-5-15-u	5031.67	4999.84	-31.83	-0.62	20	18
total-5-11-u	84.71	83.81	-0.90	-1.06	14	13
total-5-13-u	206.64	204.36	-2.28	-1.10	13	12
dated-10-15-u	97.12	84.04	-13.08	-14.43	32	19
dspam_dump_vc973	18202.99	17754.12	-448.87	-2.50	38	35
manol-pipe-c10nidw_s	919.38	919.49	0.11	0.01	10	10
TOTAL(17)	34214.88	33724.8	-490.08	-1.44	26	24

this is achieved without increasing the computation time; indeed, modest decreases were achieved.

We thank Armin Biere for many helpful email discussions.

References

1. Biere, A.: Picosat Essentials. J. Satisfiability 4, 75–97 (2008)
2. Beame, P., Kautz, H., Sabharwal, A.: Towards Understanding and Harnessing the Potential of Clause Learning. J.A.I.R. 22, 319–351 (2004)
3. Baase, S., Van Gelder, A.: Computer Algorithms: Introduction to Design and Analysis, 3rd edn. Addison-Wesley, Reading (2000)
4. Goldberg, E., Novikov, Y.: Verification of Proofs of Unsatisfiability for CNF Formulas. In: Proc. Design, Automation and Test in Europe, pp. 886–891 (2003)
5. Marques-Silva, J.P., Sakallah, K.A.: GRASP–A Search Algorithm for Propositional Satisfiability. IEEE Transactions on Computers 48, 506–521 (1999)
6. Sinz, C., Biere, A.: Extended Resolution Proofs for Conjoining BDDs. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 600–611. Springer, Heidelberg (2006)
7. Van Gelder, A.: Verifying Propositional Unsatisfiability: Pitfalls to Avoid. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 328–333. Springer, Heidelberg (2007)
8. Zhang, L., Malik, S.: Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula. In: SAT 2003, Sta. Marguerita, It. (2003)
9. Zhang, L., Malik, S.: Validating SAT Solvers Using an Independent Resolution-Based Checker. In: Proc. Design, Automation and Test in Europe (2003)
10. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In: ICCAD (November 2001)

Boundary Points and Resolution

Eugene Goldberg

eu.goldberg@gmail.com

Abstract. We use the notion of boundary points to study resolution proofs. Given a CNF formula F, a $lit(x)$-boundary point is a complete assignment falsifying only clauses of F having the same literal $lit(x)$ of variable x. A $lit(x)$-boundary point mandates a resolution on variable x. Adding the resolvent of this resolution to F eliminates this boundary point. Any resolution proof has to eventually eliminate all boundary points of F. Hence one can study resolution proofs from the viewpoint of boundary point elimination. We use equivalence checking formulas to compare proofs of their unsatisfiability built by a conflict driven SAT-solver and very short proofs tailored to these formulas. We show experimentally that in contrast to proofs generated by this SAT-solver, almost every resolution of a specialized proof eliminates a boundary point. This implies that one may use the share of resolutions eliminating boundary points as a metric for proof quality.

Keywords: SAT-solver, boundary points, resolution, proof quality.

1 Introduction

Resolution-based SAT-solvers [4,9,13,16,18,19,20] have achieved great success in numerous applications. Importantly, in many cases, the quality of resolution proofs generated by a SAT-solver matters as much as its performance. For example, in bounded model checking [5], a resolution proof is used to identify the part of the circuit relevant to a particular property [7]. In interpolation based model checking [15], the size of interpolant strongly depends on that of the resolution proof it is extracted from. In [10], we showed that high-quality tests can be obtained from a resolution proof. The number of these tests is proportional to the number of resolutions in the proof. So proof reduction means getting a more compact test set.

In this paper, we study the relation between resolution and boundary points [12]. The motivation here is as follows. To show that a CNF formula is unsatisfiable it is sufficient to eliminate all its boundary points. Resolution can be viewed as a method of boundary point elimination. Relating resolution proofs with boundary point elimination, may lead to a) better understanding of resolution proofs; b) identifying proof redundancies; c) designing SAT-solvers that generate smaller proofs.

Given a CNF formula $F(x_1,..,x_n)$, a non-satisfying complete assignment $\boldsymbol{p}$ is called a $lit(x_i)$-*boundary* point, if it falsifies only the clauses of F that have literal $lit(x_i)$. The name is due to the fact that for satisfiable formulas the set of such points contain the boundary between satisfying and unsatisfying assignments. If F is unsatisfiable, for every $lit(x_i)$-boundary point $\boldsymbol{p}$ there is a resolvent of two clauses of F on variable x_i

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 147–160, 2009.

that eliminates $\boldsymbol{p}$ (i.e. after adding such a resolvent to F, $\boldsymbol{p}$ is not a boundary point anymore). On the contrary, for a non-empty satisfiable formula F, there is always a boundary point that can not be eliminated by adding a clause implied by F.

To prove that a CNF formula F is unsatisfiable it is sufficient to eliminate all its boundary points. In the resolution proof system [3], one reaches this goal by adding to F resolvents. If formula F has a $lit(x_i)$-boundary point, a resolution proof has to have a resolution operation on variable x_i. The resolvents of a resolution proof eventually eliminate all boundary points. (A formula with an empty clause does not have any boundary points.) However, as we show experimentally many resolution operations of proofs generated by a conflict driven SAT-solver do not eliminate any boundary points (non-boundary resolutions). We use the ratio of boundary and non-boundary resolutions of a proof as a redundancy measure called SBR metric (Share of Boundary Resolutions).

To check if there is a relation between the value of SBR metric and proof quality we computed the values of this metric for two kinds of proofs for equivalence checking formulas. (These formulas describe equivalence checking of two copies of a combinational circuit.) Namely, we considered short proofs of linear size particularly tailored for equivalence checking formulas and much longer proofs generated by a SAT-solver with conflict driven learning. We showed experimentally that the share of boundary resolution operations in high-quality specialized proofs is much greater than in proofs generated by the SAT-solver. This implies that the SAT-solver's proofs may have some redundancies.

The contribution of this paper is threefold. First, we show that one can view resolution as elimination of boundary points. Second, we introduce the SBR metric that can be potentially used as a measure of proof redundancy. Third, we give some experimental results about the relation between SBR -metric and proof quality.

This paper is structured as follows. Section 2 introduces main definitions. Some properties of boundary points are given in Section 3. Section 4 views a resolution proof as a process of boundary point elimination. A class of equivalence checking formulas and their short resolution proofs are described in Section 5. Experimental results are given in Section 6. Some relevant background is recalled in Section 7. Conclusions and directions for future research are listed in Section 8.

2 Basic Definitions

Definition 1. A *literal* of a Boolean variable x_i (denoted as $lit(x_i)$) is a Boolean function of x_i. The identity function (denoted just as x_i) is called the *positive literal* of x_i. The negation function (denoted as $\sim x_i$) is called the *negative literal* of x_i.

Definition 2. A *clause* is the disjunction of literals where no two (or more) literals of the same variable can appear. A *CNF formula* is the conjunction of clauses. We will also view a CNF formula as a *set of clauses*.

Definition 3. Given a CNF formula $F(x_1,..,x_n)$, a *complete assignment* (also called a *point*) is a mapping $\{x_1,..,x_n\} \rightarrow \{0,1\}$. Given a complete assignment $\boldsymbol{p}$ and a clause C, denote by $C(\boldsymbol{p})$ the value of C when its variables are assigned by $\boldsymbol{p}$. A clause C is

satisfied (respectively *falsified*) by a complete assignment $\boldsymbol{p}$, if $C(\boldsymbol{p}) = 1$ (respectively $C(\boldsymbol{p}) = 0$).

Definition 4. Given a CNF formula F, a *satisfying assignment* $\boldsymbol{p}$ is a complete assignment satisfying every clause of F. *The satisfiability problem* (SAT) is to find a satisfying assignment for F or to prove that such an assignment does not exist.

Definition 5. Let F be a CNF formula and $\boldsymbol{p}$ be a complete assignment. Denote by $Unsat(\boldsymbol{p},F)$ the set of all clauses of F falsified by $\boldsymbol{p}$.

Definition 6. Given a CNF formula $F(x_1,..,x_n)$, a complete assignment $\boldsymbol{p}$ is called a $lit(x_i)$-*boundary point* if $Unsat(\boldsymbol{p},F)$ is not empty and every clause of $Unsat(\boldsymbol{p},F)$ contains literal $lit(x_i)$.

Example 1. Let F consist of 5 clauses: $C_1 = x_2$, $C_2 = \sim x_2 \vee x_3$, $C_3 = \sim x_1 \vee \sim x_3$, $C_4 = x_1 \vee \sim x_3$, $C_5 = \sim x_2 \vee \sim x_3$. Complete assignment $\boldsymbol{p_1}=(x_1=0,x_2=0,x_3=1)$ falsifies clauses C_1,C_4. So $Unsat(\boldsymbol{p}_1,F)=\{C_1,C_4\}$. There is no literal shared by the clauses of $Unsat(\boldsymbol{p}_1,F)$. Hence $\boldsymbol{p_1}$ is not a boundary point. On the other hand, $\boldsymbol{p_2}=(x_1=0,x_2=1,x_3=1)$ falsifies the clauses C_4,C_5 sharing literal $\sim x_3$. So $\boldsymbol{p_2}$ is a $\sim x_3$-boundary point.

3 Basic Properties of Boundary Points

In this section we give some properties of boundary points.

3.1 Basic Propositions

In this subsection, we prove the following propositions. The set of boundary points contains the boundary between satisfying and unsatisfying assignments (Proposition 1). A CNF formula without boundary points is unsatisfiable (Proposition 2). Boundary points come in pairs (Proposition 3).

Definition 7. Denote by $Bnd\_pnts(F)$ the set of all boundary points of a CNF formula F. We assume that a $lit(x_i)$-boundary point $\boldsymbol{p}$ is specified in $Bnd\_pnts(F)$ as the pair $(lit(x_i), \boldsymbol{p})$. So the same point $\boldsymbol{p}$ may be present in $Bnd\_pnts(F)$ more than once (e.g. if $\boldsymbol{p}$ is a $lit(x_i)$-boundary point and a $lit(x_j)$-boundary point at the same time).

Proposition 1. Let F be a satisfiable formula whose set of clauses is not empty. Let $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ be two complete assignments such that a) $F(\boldsymbol{p_1})=0$, $F(\boldsymbol{p_2})=1$; b) $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ are different only in the value of variable x_i. Then $\boldsymbol{p_1}$ is a $lit(x_i)$-boundary point.

Proof. Assume the contrary i.e. $Unsat(\boldsymbol{p_1},F)$ contains a clause C of F that does not have variable x_i. Then $\boldsymbol{p_2}$ falsifies C too and so $\boldsymbol{p_2}$ cannot be a satisfying assignment. A contradiction.

Proposition 1 means that the set $Bnd\_pnts(F)$ contains the boundary between satisfying and unsatisfying assignments of a satisfiable CNF formula F.

Proposition 2. Let F be a CNF formula that has at least one clause. If $Bnd\_pnts(F) = \varnothing$, then F is unsatisfiable.

Proof. Assume the contrary i.e. $Bnd\_pnts(F) = \varnothing$ and F is satisfiable. Since F is not empty, one can always find two points $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ such that $F(\boldsymbol{p_1})=0$ and $F(\boldsymbol{p_2})=1$ and that are different only in the value of one variable x_i of F. Then according to Proposition 1, $\boldsymbol{p_1}$ is a $lit(x_i)$-boundary point. A contradiction.

Proposition 3. Let $\boldsymbol{p_1}$ be a $lit(x_i)$-boundary point for a CNF formula F. Let $\boldsymbol{p_2}$ be the point obtained from $\boldsymbol{p_1}$ by changing the value of x_i. Then $\boldsymbol{p_2}$ is either a satisfying assignment or a $\sim lit(x_i)$-boundary point.

Proof. Reformulating the proposition, one needs to show that $Unsat(\boldsymbol{p_2},F)$ is either empty or contains only clauses with $\sim lit(x_i)$. Assume that contrary, i.e. $Unsat(\boldsymbol{p_2},F)$ contains a clause C with no literal of x_i. (All clauses with $lit(x_i)$ are satisfied by $\boldsymbol{p_2}$.) Then C is falsified by $\boldsymbol{p_1}$ too and so $\boldsymbol{p_1}$ is not a $lit(x_i)$-boundary point. A contradiction.

Definition 8. Proposition 3 means that for unsatisfiable formulas every x_i-boundary point has the corresponding $\sim x_i$-boundary point (and vice versa). We will call such a pair of points *twin boundary points in variable* x_i.

Example 2. The point $\boldsymbol{p_2}= (x_1=0,x_2=1,x_3=1)$ of Example 1 is an $\sim x_3$-boundary point. The point $\boldsymbol{p_3}=(x_1=0,x_2=1, x_3=0)$ obtained from $\boldsymbol{p_2}$ by flipping the value of x_3 falsifies only clause $C_2= \sim x_2 \vee x_3$. So $\boldsymbol{p_3}$ is an x_3-boundary point.

3.2 Elimination of Boundary Points by Adding Resolvents

In this subsection, we prove the following propositions. Clauses of a CNF formula F falsified by twin boundary points can be resolved (Proposition 4). Adding such a resolvent to F eliminates these boundary points (Proposition 5). Adding the resolvents of a resolution proof eventually eliminates all boundary points (Proposition 6). A $lit(x_i)$-boundary point can be eliminated only by a resolution on variable x_i (Proposition 7). If formula F has a $lit(x_i)$-boundary point, any resolution proof that F is unsatisfiable has a resolution on variable x_i. (Proposition 8).

Definition 9. Let C_1 and C_2 be two clauses that have opposite literals of variable x_i (and no opposite literals of any other variable). The *resolvent* C of C_1 and C_2 is the clause consisting of all the literals of C_1 and C_2 but the literals of x_i. The clause C is said to be obtained by *a resolution operation* on variable x_i. C_1 and C_2 are called the *parent clauses*.

Proposition 4. Let $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ be twin boundary points of a CNF formula F in variable x_i. Let C_1 and C_2 be two arbitrary clauses falsified by $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ respectively. Then a) C_1 , C_2 can be resolved on variable x_i; b) $C(\boldsymbol{p_1}) = 0$, $C(\boldsymbol{p_2}) = 0$ where C is the resolvent of C_1 and C_2.

Proof. Since $C_1(\boldsymbol{p_1})=0$, $C_2(\boldsymbol{p_2})=0$ and $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ are twin boundary points in x_i, C_1 and C_2 have opposite literals of variable x_i. Since $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ are different only in the value

of x_i, clauses C_1 and C_2 can not contain opposite literals of a variable other than x_i. (Otherwise, $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ had to be different in values of at least 2 variables.) Since $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ are different only in the value of x_i, they both set to 0 all the literals of C_1 and C_2 but literals of x_i. So the resolvent C of C_1 and C_2 is falsified by $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$.

Example 3. Points $\boldsymbol{p_2}$= (x_1=0, x_2=1, x_3=1) and $\boldsymbol{p_3}$=(x_1=0, x_2=1, x_3=0) from Examples 1 and 2 are twin boundary points in variable x_3. $Unsat(\boldsymbol{p_2}, F)$={$C_4$,$C_5$} and $Unsat(\boldsymbol{p_3}, F)$ ={C_2}. For example, $C_4 = x_1 \vee \sim x_3$, can be resolved with C_2=$\sim x_2 \vee x_3$ on variable x_3. Their resolvent C= $x_1 \vee \sim x_2$ is falsified by both $\boldsymbol{p_2}$ and $\boldsymbol{p_3}$.

Proposition 5. Let $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ be twin boundary points in variable x_i and C_1 and C_2 be clauses falsified by $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ respectively Then adding the resolvent C of C_1 and C_2 to F *eliminates* the boundary points $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$. That is pairs (x_i,$\boldsymbol{p_1}$) and ($\sim x_i$,$\boldsymbol{p_2}$) are not in the set $Bnd\_pnts(F \wedge C)$ (here we assume that $\boldsymbol{p_1}$ is an x_i-boundary point and $\boldsymbol{p_2}$ is a $\sim x_i$-boundary point of F).

Proof. According to Proposition 4, any clauses C_1 and C_2 falsified by $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ respectively can be resolved in x_i and $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ falsify the resolvent C of C_1 and C_2. Since clause C does not have a literal of x_i , $\boldsymbol{p_1}$ is not an x_i-boundary point and $\boldsymbol{p_2}$ is not a $\sim x_i$-boundary point of $F \wedge C$.

Proposition 6. If a CNF formula F contains an empty clause, then $Bnd\_pnts(F) = \varnothing$.

Proof. For any complete assignment $\boldsymbol{p}$, the set $Unsat(\boldsymbol{p},F)$ contains the empty clause of F. So $\boldsymbol{p}$ can not be a $lit(x_i)$-boundary point.

Proposition 6 works only in one direction, i.e. if $Bnd\_pnts(F) = \varnothing$, it does not mean that F contains an empty clause. Proposition 6 implies that, given an unsatisfiable formula F for which $Bnd\_pnts(F)$ is not empty , the resovlents of any resolution proof of unsatisfiability of F eventually eliminate all the boundary points.

Proposition 7. Let F be a CNF formula and $\boldsymbol{p}$ be a $lit(x_i)$-boundary point of F. Let C be the resolvent of clauses C_1 and C_2 of F that eliminates $\boldsymbol{p}$ (i.e. ($lit(x_i)$,$\boldsymbol{p}$) is not in $Bnd\_pnts(F \wedge C)$). Then C is obtained by resolution on variable x_i. In other words, a $lit(x_i)$-boundary point can be eliminated only by adding to F a resolvent on variable x_i.

Proof. Assume the contrary i.e. adding C to F eliminates $\boldsymbol{p}$ and C is obtained by resolving C_1 and C_2 on variable x_j, $j \neq i$. Since C eliminates $\boldsymbol{p}$ as a $lit(x_i)$-boundary point, it is falsified by $\boldsymbol{p}$ and does not contain $lit(x_i)$. This means that neither C_1 nor C_2 contain variable x_i. Since C is falsified by $\boldsymbol{p}$, one of the parent clauses, say clause C_1, is falsified by $\boldsymbol{p}$ too. Since C_1 does not contain literal $lit(x_i)$, $\boldsymbol{p}$ is not a $lit(x_i)$-boundary point of F. A contradiction.

Proposition 8. Let $\boldsymbol{p}$ be a $lit(x_i)$-boundary point of a CNF formula F. Then any resolution derivation of an empty clause from F has to contain a resolution operation on variable x_i.

Proof. According to Proposition 6, every boundary point of F is eventually eliminated in a resolution proof. According to Proposition 7, a $lit(x_i)$-boundary point can be eliminated only by adding to F a clause produced by resolution on variable x_i.

3.3 Boundary Points and Clause Redundancy

In this subsection, we show that redundant clauses (e.g. conflict clauses) can be used in resolutions as parent clauses to eliminate boundary points.

Definition 10. A clause C of a CNF formula F is called *redundant* if $F \setminus \{C\} \rightarrow C$.

Proposition 9. Let C be a clause of a CNF formula F. Let $lit(x_i)$ be a literal of C. Suppose that no $lit(x_i)$-boundary point of F falsifies clause C. Then C is redundant.

Proof. Assume the contrary, i.e. C is not redundant. Then there is an assignment $\boldsymbol{p}$ such that C is falsified and all the other clauses of F are satisfied. Then $\boldsymbol{p}$ is a $lit(x_i)$-boundary point. A contradiction.

Importantly, Proposition 9 works only in one direction. That is the fact that a clause C is redundant in F does not mean that no boundary point of F falsifies C. Let CNF formula $F(x_1,x_2)$ consist of four clauses: $\sim x_1$, x_1, $x_1 \vee \sim x_2$, $x_1 \vee x_2$. Although the clause x_1 is redundant in F, $\boldsymbol{p} = (x_1=0,\ x_2=0)$ is an x_1-boundary point falsifying x_1 (and $x_1 \vee x_2$). The resolvent of clauses x_1 and $\sim x_1$ eliminates $\boldsymbol{p}$ as a boundary point.

4 Resolution Proofs and Boundary Points

In this section, we view construction of a resolution proof as a process of boundary point elimination and give a metric for measuring potential proof redundancy.

4.1 Resolution Proof as Boundary Point Elimination

First, we define the notion of a resolution proof [3] and a boundary resolution.

Definition 11. Let F be an unsatisfiable formula. Let $R_1,\ldots,R_k$ be a set of clauses such that a) each clause R_i is obtained by resolution operation where a parent clause is either a clause of F or the resolvent of a previous resolution operation; b) clauses R_i are numbered in the order they are derived; c) R_k is an empty clause; Then the set of resolutions that produced the resolvents $R_1,..,R_k$ is called a *resolution proof*. We assume that this proof is *irredundant* i.e. removal of any non-empty subset of these k resolvents breaks condition a).

Definition 12. Let $R_1,..,R_k$ be the set of resolvents forming a resolution proof that a CNF formula F is unsatisfiable. Denote by F_i the CNF formula that is equal to F for i=1 and to $F \cup \{R_1,..,R_{i-1}\}$ for $i = 2,\ldots,k$.. In other words, F_i consists of the initial clauses of F and first i-1 resolvents. We will say that the i-th resolution (i.e. one that produces resolvent R_i) is *non-boundary* if $Bnd\_pnts(F_i) = Bnd\_pnts(F_{i+1})$. Otherwise (i.e. if $Bnd\_pnts(F_i) \supset Bnd\_pnts(F_{i+1})$, because adding a clause can not *create* a

boundary point), *i*-th resolution is called *boundary*. So a resolution operation is boundary if adding R_i to F_i eliminates a boundary point.

In the previous section, we showed that eventually all the boundary points of a CNF formula F are removed by resolvents. Importantly, a $lit(x_i)$-boundary point mandates a resolution on variable x_i. Besides, as we showed in Section 3.3. even redundant clauses can be used to produce new resolvents eliminating boundary points. It is important because, all clauses derived by resolution (in particular conflict clauses generated by modern SAT-solvers) are redundant. So the derived clauses are as good for boundary point elimination as the original ones.

A natural question arises about the role of non-boundary resolutions. When answering this question it makes sense to separate redundant and irredundant formulas. (A CNF formula F is said to be *irredundant* if no clause of F is redundant, see Definition 10) For a redundant formula, one may have to use non-boundary resolutions. (In particular, a heavily redundant formula may not have boundary points at all. In such a case, every resolution operation is non-boundary.) For irredundant formulas the situation is different.

Proposition 10. Let F be an irredundant formula of m clauses. Then F has at least d boundary points where d is the total number of literals in the clauses of F.

Proof. Let C be a clause of F. Then there is a complete assignment $\boldsymbol{p}$ falsifying C and satisfying the clauses of $F \setminus \{C\}$. This assignment is a $lit(x_i)$-boundary point where $lit(x_i)$ is a literal of C.

As far as irredundant formulas are concerned some natural questions arise. Given an irredundant formula, can one always build a resolution proof using only boundary resolutions? If so, what is the relation between such a limited proof system and general resolution? Can general resolution produce proofs shorter than in this limited proof system?

We do not answer the questions above theoretically. Instead we introduce a metric measuring the Share of Boundary Resolutions (SBR-metric) to check if proof quality depends on the value of SBR metric. (This value is computed as the percent of boundary resolutions of a proof). In Section 6, we give some experimental evidence that the value of SBR metric for short specialized proofs for equivalence checking formulas is much higher than for proofs generated by a SAT-solver with conflict driven learning.

4.2 SBR Metric and Proof Redundancy

Although we do not know the nature of non-boundary resolutions we still can argue that the low value of SBR metric may mean some proof redundancy. The reason is that such a redundancy indeed leads to the appearance of non-boundary resolutions. We give two examples of that below.

Not sharing resolutions of conflict clause derivations. In a typical SAT-solver with conflict-driven learning, the only type of clauses learned are conflict clauses (this applies to the Sat-solver DMRP-SAT that we used in our experiments). On the other

hand, a conflict clause is the result of many resolution operations. The intermediate resolvents of these operations are usually discarded. It may be the case though that for two conflict clauses C_1 and C_2 resolution proofs of their derivation share some intermediate resolvents. When the value of SBR metric is computed, *all resolvents* are taken into consideration. This lack of sharing resolutions used in conflict clause derivations would lead to appearance of non-boundary resolutions (a resolution deriving a clause produced earlier by some other resolution can not eliminate a boundary point).

Appearance of unsatisfiable subformulas. Even if the initial unsatisfiable CNF formula F to be solved is irredundant, an unsatisfiable subformula of F inevitably appears due to the addition of new clauses. (In particular, one can view an empty clause as an unsatisfiable subformula of the final CNF formula.) Let F_1 be an unsatisfiable subformula of F. Let $Vars(F)$ and $Vars(F_1)$ denote the sets of variables of F and F_1. Then no $lit(x_i)$-boundary point exists if x_i is in $Vars(F) \setminus Vars(F_1)$. (The set of clauses falsified by $\boldsymbol{p}$ contains at least one clause of F_1 and such a clause does not have a literal of x_i.) So any resolution on a variable of $Vars(F) \setminus Vars(F_1)$ is non-boundary.

The appearance of unsatisfiable subformulas may lead to increasing the share of non-boundary resolutions in the final proof. For example, instead of deriving an empty clause from the clauses of F_1, the SAT-solver may first derive some clauses having variables of $Vars(F_1)$ from clauses of $F \setminus F_1$. It is possible since clauses of $F \setminus F_1$ may contain variables of $Vars(F_1)$. When deriving such clauses the SAT-solver may use (non-boundary) resolutions on variables of $Vars(F) \setminus Vars(F_1)$, which leads to redundancy of the final proof.

5 Equivalence Checking Formulas

In this section, we introduce the formulas we use in the experimental part of this paper. These are the formulas that describe equivalence checking of two copies of a combinational circuit. In Subsection 5.1 we show how such formulas are constructed. In Subsection 5.2. we describe how short proofs of unsatisfiability particularly tailored for equivalence checking formulas can be built.

5.1 Building Equivalence Checking Formulas

Let N and N^* be two single-output combinational circuits. To check their functional equivalence one constructs a circuit called a *miter* (we denote it as $Miter(N,N^*)$). It is a circuit that is satisfiable (i.e. its output can be set to 1) if and only if N and N^* are not functionally equivalent. (N and N^* are not functionally equivalent if there is an input assignment for which N and N^* produce different output values.) Then a CNF formula F_{Miter} is generated that is satisfiable if and only if $Miter(N,N^*)$ is satisfiable. In our experiments we built a miter of two *identical copies* of the same circuit. In such a case $Miter(N,N^*)$ is always unsatisfiable and so is CNF formula F_{Miter}.

Example 4. Figure 1 shows the miter of copies N and N^* of the same circuit. Here g_1, g_1^* are OR gates, g_2, g_2^* are AND gates and h is an XOR gate (implementing modulo-2 sum). Note that N and N^* have the same set of input variables but different

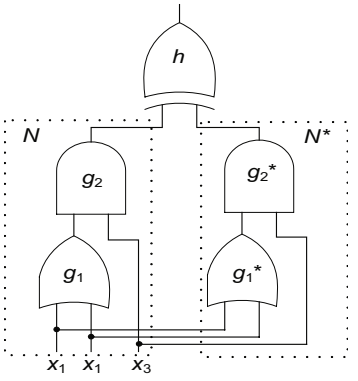


Fig. 1. Circuit *Miter*(*N*, *N**)

intermediate and output variables. Since $g_2 \oplus g_2^*$ evaluates to 1 if and only if $g_2 \neq g_2^*$, and N and N^* are functionally equivalent, the circuit *Miter*(N,N^*) evaluates only to 0.

A CNF formula F_{Miter} whose satisfiability is equivalent to that of *Miter*(N,N^*) is formed as $F_N \wedge F_{N^*} \wedge F_{\text{xor}} \wedge h$. Here F_N and F_{N^*} are formulas specifying the functionality of N and N^* respectively. The formula F_{xor} specifies the functionality of the XOR gate h and the unit clause h forces the output of *Miter*(N,N^*) to be set to 1.

Since, in our case, the miter evaluates only to 0, the formula F_M is unsatisfiable.

Formulas F_N and F_{N^*} are formed as the conjunction of subformulas describing the gates of N and N^*. For instance, $F_N = F_{g1} \wedge F_{g2}$ where, for example, $F_{g1} = (x_1 \vee x_2 \vee \sim g_1) \wedge (\sim x_1 \vee g_1) \wedge (\sim x_2 \vee g_1)$ specifies the functionality of an OR gate. Each clause of F_{g1} rules out some inconsistent assignments to the variables of gate g_1. For example, the clause $(x_1 \vee x_2 \vee \sim g_1)$ rules out the assignment x_1=0, x_2=0, g_1=1.

5.2 Short Proofs for Equivalence Checking Formulas

For a CNF formula F_{Miter} describing equivalence checking of two copies N,N^* of the same circuit, there is a short resolution proof that F_{Miter} is unsatisfiable. This proof is linear in the number of gates in N and N^*. The idea of this proof is as follows. For every pair g_i, g_i^* of the corresponding gates of N and N^* , the clauses of CNF formula $Eq(g_i,g_i^*)$ specifying the equivalence of variables g_i, g_i^* are derived. Here $Eq(g_i,g_i^*)$ is equal to $(\sim g_i \vee g_i^*) \wedge (g_i \vee \sim g_i^*)$. These clauses are derived according to topological levels of gates g_i,g_i^* in *Miter*(N,N^*). (The topological level of a gate g_i is the length of the longest path from an input to gate g_i measured in the number of gates on this path.) First, clauses of $Eq(g_i,g_i^*)$ are derived for all pairs of gates g_i,g_i^* of topological level 1. Then using previously derived $Eq(g_i,g_i^*)$, same clauses are derived for the pairs of gates g_j,g_j^* of topological level 2 and so on.

Eventually, the clauses of $Eq(g_s,g_s^*)$ relating the output variables g_s, g_s^* of N and N^* are derived. Resolving the clauses of $Eq(g_s,g_s^*)$ and the clauses describing the XOR gate, the clause $\sim h$ is derived. Resolution of $\sim h$ and the unit clause h of F_{Miter} produces an empty clause.

Example 5. Let us explain the construction of the proof using the CNF F_{Miter} from Example 4. Gates g_1,g_1^* have topological level 1 in *Miter*(N,N^*). So first, the clauses of $Eq(g_1,g_1^*)$ are obtained. They are derived from the CNF formulas F_{g1} and F_{g1^*} describing gates g_1 and g_1^*. That the clauses of $Eq(g_1,g_1^*)$ can be derived from $F_{g1} \wedge F_{g1}^*$ just follows from the completeness of resolution and the fact that $Eq(g_1,g_1^*)$ is implied by the CNF formula $F_{g1} \wedge F_{g1^*}$. (This implication is due to the fact that F_{g1} and F_{g1}^* describe two functionally equivalent gates with the same set of input variables). More specifically, the clause $\sim g_1 \vee g_1^*$ is obtained by resolving the clause $x_1 \vee x_2 \vee \sim g_1$ of

F_{g1} with the clause $\sim x_1 \vee g_{1*}$ of F_{g1}* and then resolving the resolvent with the clause $\sim x_2 \vee g_1$* of F_{g1*}. In a similar manner, the clause $g_1 \vee \sim g_1$* is derived by resolving the clause $x_1 \vee x_2 \vee \sim g_1$* of F_{g1*} with clauses $\sim x_1 \vee g_1$ and $\sim x_2 \vee g_1$ of F_{g1}.

Then the clauses of $Eq(g_2,g_2$*) are derived (gates g_2,g_2* have topological level 2). $Eq(g_2,g_2$*) is implied by $F_{g2} \wedge F_{g2}$* $\wedge Eq(g_1,g_1$*) . Indeed, g_2 and g_2* are functionally equivalent gates that have the same input variable x_3. The other input variables g_1 and g_1* are identical too due to the presence of $Eq(g_1,g_1$*). So the clauses of $Eq(g_2,g_2$*) can be derived from clauses of $F_{g2} \wedge F_{g2}$* $\wedge Eq(g_1,g_1$*) by resolution. Then the clause $\sim h$ is derived as implied by $F_{xor} \wedge Eq(g_2,g_{2*})$ (an XOR gate produces output 0 when its input variables have equal values). Resolution of h and $\sim h$ produces an empty clause.

6 Experimental Results

The goal of experiments was to compare the values of SBR metric for two kinds of proofs of different quality. In the experiments we used formulas describing the equivalence checking of two copies of combinational circuits. The reason for using such formulas is that one can easily generate high-quality specialized proofs of their unsatisfiability (see Section 5). In the experiments we compared these short proofs with ones generated by the SAT-solver DMRP-SAT [11].

We performed the experiments on a Pentium-4 PC with clock frequency of 3 GHz. The CNF formulas and proofs of both types we used in the experiments can be downloaded from http://eigold.tripod.com/exper_sat_2009.tar.gz. The time limit in all experiments was set to 1 hour.

Given a resolution proof R of k resolutions that a CNF formula F is unsatisfiable, computing the value of SBR metric of R reduces to k SAT-checks. In our experiments these SAT-checks were performed by a version of DMRP-SAT. Let F_i be the CNF formula $F \cup \{R_1,..,R_{i-1}\}$ where $R_1,..,R_{i-1}$ are the resolvents generated in the first i-1 resolutions. Let C_1 and C_2 be the clauses of F_i that are the parent clauses of the resolvent R_i. Let C_1 and C_2 be resolved on variable x_j. Assume that C_1 contains the positive literal of x_j. Checking if i-the resolution eliminates an x_j-boundary point can be performed as follows. First, all the clauses with a literal of x_j are removed from F_i. Then one adds to F_i the unit clauses that force the assignments setting all the literals of C_1 and all the literals of C_2 but the literal $\sim x_j$ to 0. Denote the resulting CNF formula by G_i.

If G_i is satisfiable then there is a complete assignment $\boldsymbol{p}$ that is falsified by C_1 and maybe by some other clauses with literal x_j. So $\boldsymbol{p}$ is an x_j-boundary point of F_i. Since $\boldsymbol{p}$ falsifies all the literals of C_2 but $\sim x_j$, it is falsified by the resolvent of C_1 and C_2. So the satisfiability of G_i means that i-th resolution eliminates $\boldsymbol{p}$ and so this resolution is boundary. If G_i is unsatisfiable, then no x_j-boundary point is eliminated by i-th resolution. All boundary points come in pairs (see Proposition 3). So no $\sim x_j$-boundary point is eliminated by i-th resolution either. Hence the unsatisfiability of G_i means that the i-th resolution is non-boundary.

Table 1. shows the value of SBR metric for the short specialized proofs. The first column gives the name of the benchmark circuit whose self-equivalence is described by the corresponding CNF formula. The size of this CNF formula is given in the second and third columns. The fourth column of Table 1 gives the size of the proof

(in the number of resolutions). The fifth column shows the value of SBR metric and the last column of Table 1 gives the run time of computing this value. These run times can be significantly improved if one uses a faster SAT-solver and tunes it to solving the particular problem of computing the value of SBR metric. (For example, one can try to share conflict clauses learned in different SAT-checks.)

Looking at Table 1 one can conclude that the specialized proofs have a very high value of SBR-metric (almost every resolution operation eliminates a boundary point). The only exception is the *dalu* formula (84%). The fact that the value of SBR metric for *dalu* and some other formulas is different from 100% is probably due to the fact that the corresponding circuits have some redundancies. Such redundancies would lead to redundancy of CNF formulas specifying the corresponding miters, which would lower the value of SBR metric.

Table 1. Computing value of SBR metric for short specialized proofs

Name	#vars	#clauses	#resolu-tions	#boundary resol. %	run time (s)
c432	480	1,333	1,569	95	10.2
9symml	480	1,413	1,436	100	4.5
c880	807	2,264	2.469	100	24.6
alu4	2,369	7,066	8.229	96	270
c3540	2,625	7,746	9.241	97	1,743
x1	4,381	12,991	12,885	97	2,890
dalu	4,714	13,916	15,593	84	2,202
c6288	4,771	14,278	17,925	100	2,462

Table 2. Indirect comparison of proofs generated by Minisat and DMRP-SAT

Name	Minisat (v1.14)		DMRP-SAT	
	#confl. clauses	run time (sec.)	#confl. clauses	run time (sec.)
c432	809	0.05	374	0.08
9symml	259	0.03	275	0.08
c880	6,000	0.29	1,309	0.27
alu4	2,355	0.33	1,320	1.20
c3540	22,214	2.65	10,021	7.75
x1	3,294	0.45	765	1.06
dalu	4,402	0.94	3,351	4.39
c6288	*	> 3,600	*	> 3,600

Table 2 is meant to show that the proofs generated by DMRP-SAT have high quality (for a SAT-solver with conflict driven learning). Here we compare the proofs generated by Minisat (version 1.14) and DMRP-SAT in the number of conflict clauses. (Minisat does not generate proofs so we could not compare the actual proof

sizes). Although this is an indirect comparison, it gives an idea of the quality of proofs generated by DMRP-SAT. For self-equivalence of a 16-bit multiplier (instance C6288), neither SAT-solver finished the formula within the time limit.

The values of SBR-metric for the proofs generated by DMRP-SAT are given in Table 3. The second column gives the size of resolution proofs generated by DMRP-SAT. When computing the size of these proofs we removed the obvious redundancies. Namely, the derivation of the conflict clauses that did not contribute to the derivation of an empty clause was ignored. The third column shows the value of SBR metric and the last column gives the run time of computing this value. In the case the computation did not finish within the time limit, the number in parentheses shows the percent of the resolution operations processed before the computation terminated.

Table 3 shows that the size of the proofs generated by DMRP-SAT is much larger than that of specialized proofs (Table 1, fourth column). The only exception is the instance *x1* where the two kinds of proofs have comparable size. Interestingly, *x1* is the instance with the highest value of SBR metric (88%) among DMRP-SAT proofs. For the rest of the formulas the value of SBR metric is much smaller. Importantly, the value of SBR metric we give for the formulas for which computation was terminated due to exceeding the time limit is higher than it should be. Typically, the later a resolution occurs in a resolution proof, the more likely it is that this resolution is non-boundary. So the early termination of SBR metric computation ignored resolutions with the highest chances to be non-boundary.

Table 3. Computing value of SBR metric for proofs generated by DMRP-SAT

Name	#resolutions	#boundary resol. %	run time (s) (% of proof finished)
c432	7,655	37	48
9symml	5,317	57	23
c880	24,478	37	503
alu4	98,600	36	>3,600 (49%)
c3540	347,264	54	>3,600 (6%)
x1	16,259	88	864
dalu	119,553	40	>3,600 (33.4%)

Summing up, one can conclude that for the formulas we considered in experiments, the proofs of poorer quality (generated by DMRP-SAT) have lower values of SBR metric. It remains to be seen though whether it means that these proofs are redundant in some way and so can be optimized.

7 Some Background

The notion of boundary points was introduced in [12] where they were called essential points. (We decided to switch to the term "boundary point" as more precise.) Boundary points were used in [12] to help a SAT-solver prune the search space. If the subspace x_i=0 does not contain a satisfying assignment or an x_i-boundary point, one can claim that the symmetric subspace x_i=1 can not contain a satisfying

assignment either (due to Proposition 3). The same idea of search pruning was independently described in [17] and implemented in the SAT-solver Jerusat. The ideas of search pruning introduced in [12] were further developed in [2].

In [14] we formulate two proof systems meant for exploring the 1-neighborhood of clauses of the formula to be solved. The union of the 1-neighborhhoods of these clauses is essentially a superset approximation of the set of boundary points. To prove that a formula is unsatisfiable it is sufficient to eliminate all boundary points (Proposition 2). The proof systems of [14] show that one can eliminate all boundary points without generation of an empty clause. So resolution can be viewed as a special case of boundary point elimination.

The results of the present paper can also be considered as an approach to improving automatizability of resolution [6]. General resolution is most likely non-automatizable [1]. This means that finding short proofs can not be done efficiently in general resolution. A natural way to mitigate this problem is to look for restricted versions of general resolution that are "more automatizable" i.e. that facilitate finding good proofs. Intuitively, boundary resolutions is a tiny part of the set of all possible resolutions. So the restriction of resolutions to boundary ones can be viewed as a way to make it easier to find good proofs (assuming that such a restriction does not kill *all* high-quality proofs.)

8 Conclusions and Directions for Future Research

We show that a resolution proof can be viewed as the process of boundary point elimination. We introduce the SBR metric that is the percent of resolutions of the proof that eliminate boundary points (boundary resolutions). This metric can be used for estimating proof redundancy. We experimentally show that short specialized proofs for equivalence checking formulas have high values of SBR metric. On the other hand, values of this metric for proofs generated by a SAT-solver with conflict driven learning are low. This implies that the proofs generated by this SAT-solver may have some redundancies.

The idea of treating resolution as boundary proof elimination has many interesting directions for research. Here are a few of them.

1) Testing further the conjecture that SBR metric relates to proof redundancy.
2) Proving completeness of resolution performing only boundary resolutions for irredundant CNF formulas
3) Answering the question about the nature of non-boundary resolutions. In particular, is the resolution proof system where only boundary resolutions are allowed less powerful than general resolution?
4) Designing SAT-solvers that generate proofs with high value of SBR metric.

References

1. Alekhnovich, M., Razborov, A.: Resolution Is Not Automatizable Unless W[P] Is Tractable. SIAM J. Comput. 38(4), 1347–1363 (2008)
2. Babic, D., Bihgham, J., Hu, A.: Efficient SAT solving: beyond supercubes. In: DAC 2005, pp. 744–749 (2005)

3. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) ch. 2, vol. 1, pp. 19–99. Elsevier Science Publ., Amsterdam (2001)
4. Bierre, A.: PicoSAT essentials. JSAT 4, 75–97 (2008)
5. Bierre, A., Cimatti, A., Clarke, F.M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: DAC 1999, pp. 317–320 (1999)
6. Bonet, M.L., Pitassi, T., Raz, R.: On Interpolation and Automatization for Frege Systems. SIAM J. Comput. 29(6), 1939–1967 (2000)
7. Clarke, E., Gupta, A., Strichman, O.: SAT-based counterexample-guided abstraction refinement. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23(7), 1113–1123 (2004)
8. Davis, M., Longemann, G., Loveland, D.: A Machine program for theorem proving. Communications of the ACM 5, 394–397 (1962)
9. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
10. Goldberg, E.: On Bridging Simulation and Formal Verification. In: VMCAI-2008, pp. 127–141 (2008)
11. Goldberg, E.: A Decision-Making Procedure for Resolution-Based SAT-Solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 119–132. Springer, Heidelberg (2008)
12. Goldberg, E., Prasad, M., Brayton, R.: Using Problem Symmetry in Search Based Satisfiability Algorithms. In: DATE 2002, pp. 134–141 (2002)
13. Goldberg, E., Novikov, Y.: BerkMin: a Fast and Robust SAT-Solver. In: DATE 2002, pp. 142–149 (2002)
14. Goldberg, E.: Proving unsatisfiability of CNFs locally. J. Autom. Reasoning 28(5), 417–434 (2002)
15. McMillan, K.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
16. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC 2001, pp. 530–535 (2001)
17. Nadel, A.: Backtrack search algorithms for propositional logic satisfiability: review and innovations. Master Thesis, the Hebrew University (2002)
18. The siege sat-solver, `http://www.cs.sfu.ca/~cl/software/siege/`
19. Silva, J., Sakallah, K.: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Transactions of Computers 48, 506–521 (1999)
20. Zhang, H.: SATO: An efficient propositional prover. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 272–275. Springer, Heidelberg (1997)

Sequential Encodings from Max-CSP into Partial Max-SAT*

Josep Argelich[1], Alba Cabiscol[2], Inês Lynce[3], and Felip Manyà[4]

[1] INESC-ID, Lisbon, Portugal
[2] Computer Science Department, Universitat de Lleida, Spain
[3] IST/INESC-ID, and Technical University of Lisbon, Portugal
[4] Artificial Intelligence Research Institute (IIIA, CSIC), Spain

Abstract. We define new encodings from Max-CSP into Partial Max-SAT which are obtained by modelling the at-most-one condition with the sequential SAT encoding of the cardinality constraint $\leq 1(x_1, \ldots, x_n)$. They have fewer clauses than the existing encodings, and the experimental results indicate that they have a better performance profile.

1 Introduction

We describe, following our previous results in [2,3,4], novel encodings from Max-CSP into Partial Max-SAT. In [2,3], we defined a new encoding from CSP into SAT, called *minimal support encoding*, and defined the extensions from Max-CSP into Partial Max-SAT of the *direct* and *support* encodings from CSP into SAT, as well as the extension of the minimal support encoding. The experimental results for Partial Max-SAT provide evidence that, in general, the minimal support encoding outperforms the other encodings on both pure random [2,3] and more structured, realistic instances [4]. In the sequel, when we say direct, support and minimal support encodings we refer to the corresponding encodings from Max-CSP into Partial Max-SAT. We also refer to them as the *standard encodings*.

Recently [4], we have defined new variants of the standard encodings, called *regular direct*, *regular support* and *regular minimal support* encodings. They are obtained by modelling the at-least-one (ALO) and at-most-one (AMO) conditions of the corresponding standard encodings using a regular signed encoding [1]. This way, we get encodings with a more compact set of hard clauses, but we need to introduce auxiliary variables. Fortunately, it is sufficient to limit branching to non-auxiliary variables [4]. From a practical point of view, the regular encodings usually outperform the corresponding standard encodings.

In this paper we define new encodings —*sequential direct*, *sequential support* and *sequential minimal support* —, which are obtained by modelling the ALO

* Research funded by European project Mancoosi (FP7-ICT-214898), FCT projects Bsolo (PTDC/EIA/76572/2006) and SHIPs (PTDC/EIA/64164/2006), and the *Ministerio de Ciencia e Innovación* projects CONSOLIDER CSD2007-0022, INGENIO 2010, TIN2006-15662-C02-02, and TIN2007-68005-C04-04.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 161–166, 2009.

condition as in the standard encoding, and the AMO condition with the sequential SAT encoding of the cardinality constraint $\leq 1(x_1, \ldots, x_n)$ [6]. They have fewer clauses than the existing encodings, and the experimental results indicate that they have a better performance profile. In our experiments we solve both pure random and more structured, realistic instances. We refer to [2,3,4] for basic definitions of Max-SAT and Max-CSP.

2 Encodings from Max-CSP into Partial Max-SAT

2.1 Standard Encodings

We associate a Boolean variable x_i with each value i of the CSP variable X. If X has a domain $d(X)$ of size m, the *ALO* clause of X is $x_1 \vee \cdots \vee x_m$, and ensures that X is given a value. The *AMO* clauses are the set of clauses $\{\overline{x}_i \vee \overline{x}_j | i, j \in d(X), i < j\}$, and ensure that X takes no more than one value.

Definition 1. *The* direct encoding *(*`dir`*) of a Max-CSP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is the Partial Max-SAT instance that contains as hard clauses the above ALO and AMO clauses for every CSP variable in $\mathcal{X}$, and a soft clause $\overline{x}_i \vee \overline{y}_j$ for every nogood $(X = i, Y = j)$ of every constraint of $\mathcal{C}$ with scope $\{X, Y\}$.*

In the *support encoding* from CSP into SAT, besides the ALO and AMO clauses, there are clauses that encode the *support* for a value instead of encoding conflicts. The support for a value j of a CSP variable X across a binary constraint with scope $\{X, Y\}$ is the set of values of Y which allow $X = j$. If $v_1, v_2, \ldots, v_k$ are the supporting values of variable Y for $X = j$, we add the clause $\overline{x}_j \vee y_{v_1} \vee y_{v_2} \vee \cdots \vee y_{v_k}$ (called *support* clause). There is one support clause for each pair of variables X, Y involved in a constraint, and for each value in the domain of X. In the standard support encoding, a clause in each direction is used: one for the pair X, Y and one for Y, X [7].

In [2], we defined the *minimal support encoding*: it is like the support encoding except for the fact that, for every constraint C_k with scope $\{X, Y\}$, we only add either the support clauses for all the domain values of the CSP variable X or the support clauses for all the domain values of the CSP variable Y.

Definition 2. *The* minimal support encoding *of a Max-CSP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is the Partial Max-SAT instance that contains as hard clauses the corresponding ALO and AMO clauses for every CSP variable in $\mathcal{X}$, and as soft clauses the support clauses of the minimal support encoding from CSP into SAT.*

The support encoding *is the Partial Max-SAT instance that contains as hard clauses the corresponding ALO and AMO clauses for every CSP variable in $\mathcal{X}$, and contains, for every constraint $C_k \in \mathcal{C}$ with scope $\{X, Y\}$, a soft clause of the form $S_{X=j} \vee c_k$ for every support clause $S_{X=j}$ encoding the support for the value j of the CSP variable X, where c_k is an auxiliary variable, and contains a soft clause of the form $S_{Y=m} \vee \overline{c}_k$ for every support clause $S_{Y=m}$ encoding the support for the value m of the CSP variable Y.*

Example 1. The direct encoding for the Max-CSP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle = \langle \{X, Y\}, \{d(X) = \{1,2,3\}, d(Y) = \{1,2,3\}\}, \{X \leq Y\} \rangle$ is as follows:

ALO	$[x_1 \vee x_2 \vee x_3]\ [y_1 \vee y_2 \vee y_3]$
AMO	$[\overline{x}_1 \vee \overline{x}_2]\ [\overline{x}_1 \vee \overline{x}_3]\ [\overline{x}_2 \vee \overline{x}_3]\ [\overline{y}_1 \vee \overline{y}_2]\ [\overline{y}_1 \vee \overline{y}_3]\ [\overline{y}_2 \vee \overline{y}_3]$
conflict clauses	$(\overline{x}_2 \vee \overline{y}_1)\quad (\overline{x}_3 \vee \overline{y}_1)\quad (\overline{x}_3 \vee \overline{y}_2)$

We get the minimal support encoding if we replace the conflict clauses with $(\overline{x}_2 \vee y_2 \vee y_3), (\overline{x}_3 \vee y_3)$, and get the support encoding if we replace the conflict clauses with $(\overline{x}_2 \vee y_2 \vee y_3 \vee c_1), (\overline{y}_1 \vee x_1 \vee \overline{c}_1), (\overline{x}_3 \vee y_3 \vee c_1), (\overline{y}_2 \vee x_1 \vee x_2 \vee \overline{c}_1)$.

In the experiments we used the support encoding (`supxy`), and two variants of the minimal support encoding (`supl` and `supc`): `supl` is the encoding containing, for each constraint, the support clauses for the variable that produces a smaller total number of literals; and `supc` is the encoding containing, for each constraint, the support clauses for the variable that produces smaller size clauses; we give a score of 16 to unit clauses, a score of 4 to binary clauses and a score of 1 to ternary clauses, and choose the variable with higher sum of scores.

2.2 Regular Encodings

The regular encodings differ in the fact that they encode the ALO and AMO conditions using a regular signed encoding [1]. To this end, for every CSP variable X, we associate a Boolean variable x_i with each value i that can be assigned to the CSP variable X in such a way that x_i is true if $X = i$. Moreover, we associate a Boolean variable $x_i^{\geq}$ with each value i of the domain of X such that $x_i^{\geq}$ is true if $X \geq i$. Then, the *regular encoding of the ALO and AMO conditions* for a variable X with $d(X) = \{1, \ldots, n\}$ is formed by the following clauses [1]:

$$\begin{array}{ll} x_n^{\geq} \rightarrow x_{n-1}^{\geq} & x_1 \leftrightarrow \overline{x}_2^{\geq} \\ x_{n-1}^{\geq} \rightarrow x_{n-2}^{\geq} & x_2 \leftrightarrow x_2^{\geq} \wedge \overline{x}_3^{\geq} \\ \cdots\cdots\cdots\cdots & \cdots\cdots\cdots\cdots \\ x_3^{\geq} \rightarrow x_2^{\geq} & x_i \leftrightarrow x_i^{\geq} \wedge \overline{x}_{i+1}^{\geq} \\ x_2^{\geq} \rightarrow x_1^{\geq} & \cdots\cdots\cdots\cdots \\ & x_{n-1} \leftrightarrow x_{n-1}^{\geq} \wedge \overline{x}_n^{\geq} \\ & x_n \leftrightarrow x_n^{\geq} \end{array} \qquad (1)$$

The clauses on the left encode the relationship among the different regular literals of a variable while the clauses on the right link the variables of the form x_i with the variables of the form $x_i^{\geq}$.

Definition 3. *The* regular direct, support, and minimal support encodings *are, respectively, the standard direct, support, and minimal support encodings from Max-CSP into Partial Max-SAT but using the regular encoding of the ALO and AMO conditions.*

In [4] we proved that when solving a Max-CSP instance with a regular encoding and a Davis-Logemann-Loveland (DLL) style branch and bound solver, if branching is performed only on non-auxiliary variables, then the solver finds an optimal solution. We assume this kind of branching in the rest of the paper.

3 Sequential Encodings

Our new encodings model the ALO condition as in the standard encoding, and the AMO condition using the following SAT encoding, based on sequential counters, of the cardinality constraint $\leq 1(x_1, \ldots, x_n)$ [6]:

$$(\overline{x}_1 \vee s_1) \wedge (\overline{x}_n \vee \overline{s}_{n-1}) \bigwedge_{1<i<n} ((\overline{x}_i \vee s_i) \wedge (\overline{s}_{i-1} \vee s_i) \wedge (\overline{x}_i \vee \overline{s}_{i-1})),$$

where s_i, $1 \leq i \leq n-1$, are auxiliary variables. We refer to such an encoding as the sequential encoding of the AMO condition.

Definition 4. *The* sequential direct, support, and minimal support encodings *are, respectively, the standard direct, support, and minimal support encodings from Max-CSP into Partial Max-SAT but using the sequential encoding of the AMO condition.*

Example 2. A *sequential minimal support encoding* for the Max-CSP problem of the CSP instance from Example 1 is formed by the following clauses:

hard clauses	$[x_1 \vee x_2 \vee x_3]$	$[y_1 \vee y_2 \vee y_3]$			
	$[\overline{x}_1 \vee s_1^x]$	$[\overline{x}_3 \vee \overline{s}_2^x]$	$[\overline{x}_2 \vee s_2^x]$	$[\overline{s}_1^x \vee s_2^x]$	$[\overline{x}_2 \vee \overline{s}_1^x]$
	$[\overline{y}_1 \vee s_1^y]$	$[\overline{y}_3 \vee \overline{s}_2^y]$	$[\overline{y}_2 \vee s_2^y]$	$[\overline{s}_1^y \vee s_2^y]$	$[\overline{y}_2 \vee \overline{s}_1^y]$
support clauses	$(\overline{x}_2 \vee y_2 \vee y_3)$	$(\overline{x}_3 \vee y_3)$			

We get the *sequential support encoding* if we replace the previous support clauses with $(\overline{x}_2 \vee y_2 \vee y_3 \vee c_1), (\overline{y}_1 \vee x_1 \vee \overline{c}_1), (\overline{x}_3 \vee y_3 \vee c_1), (\overline{y}_2 \vee x_1 \vee x_2 \vee \overline{c}_1)$. Finally, we get the *sequential direct encoding* if we replace the previous support clauses with $(\overline{x}_2 \vee \overline{y}_1), (\overline{x}_3 \vee \overline{y}_1), (\overline{x}_3 \vee \overline{y}_2)$.

In the sequential encodings, the number of clauses for modelling the ALO and AMO conditions for a CSP variable X with domain $d(X)$ is on $\mathcal{O}(d(X))$. Observe that, for large domains, there are fewer clauses in the sequential encodings than in the regular and standard encodings.

Proposition 1. *When solving a Max-CSP instance with a sequential encoding and a DLL style branch and bound solver, if branching is performed only on non-auxiliary variables, then the solver finds an optimal solution.*

4 Experimental Results

We conducted experiments on a cluster with 2 GHz AMD Opteron 248 Processors, 1 GB of memory. The benchmarks are random binary Max-CSP instances as the ones solved in [3], as well as the instances of clique trees with different constraint tightness (Kbtree 10–90) and warehouse location solved in [4], which are more structured and realistic. We used the solver WMaxSatz [5] because its code is available, and we had to modify it for implementing a branching scheme

Table 1. Comparison between branching schemes

		s-supl		s-dir		s-supc		s-supxy	
Kbtree (t)	#	nb	b	nb	b	nb	b	nb	b
10	50	0.05(50)	0.10(50)	**0.03(50)**	15.01(50)	0.05(50)	1.24(50)	34.27(50)	89.18(44)
20	50	0.78(50)	46.02(50)	**0.36(50)**	345.30(49)	0.70(50)	57.07(50)	682.66(36)	0.00(0)
30	50	3.97(50)	559.65(38)	**2.92(50)**	1440.02(2)	3.78(50)	562.56(40)	0.00(0)	0.00(0)
40	50	**20.19(50)**	1175.17(3)	31.28(50)	0.00(0)	21.92(50)	1063.29(4)	0.00(0)	0.00(0)
50	50	55.31(50)	0.00(0)	96.69(50)	0.00(0)	**48.80(50)**	0.00(0)	0.00(0)	0.00(0)
60	50	**233.63(50)**	0.00(0)	549.40(50)	0.00(0)	345.89(50)	0.00(0)	0.00(0)	0.00(0)
70	50	**586.40(44)**	0.00(0)	892.17(30)	0.00(0)	1072.85(17)	0.00(0)	0.00(0)	0.00(0)
80	50	0.00(0)	0.00(0)	**1252.77(6)**	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
90	50	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
Solved	450	**344**	141	336	101	317	144	86	44
		s-supl		s-dir		s-supc		s-supxy	
Warehouse	#	nb	b	nb	b	nb	b	nb	b
warehouse	2	0.09(1)	2.37(1)	**0.08(1)**	2.35(1)	0.09(1)	2.36(1)	0.26(1)	1.34(1)
Solved	2	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

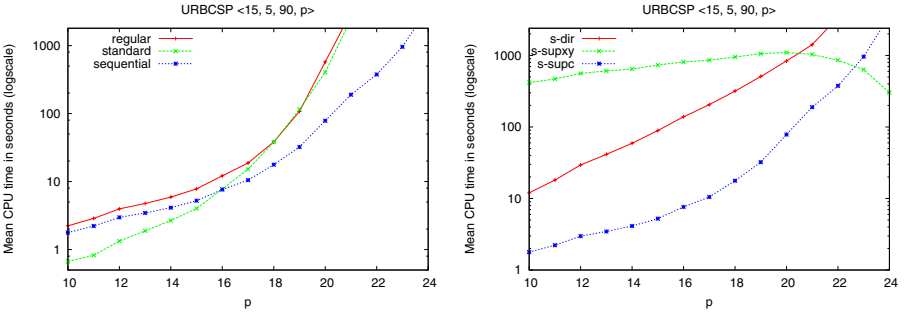


Fig. 1. Results for Random Max-CSP instances

that ignores auxiliary variables. The sequential versions of the encodings `dir`, `supc`, `supl` and `supxy` are denoted by `s-dir`, `s-supc`, `s-supl` and `s-supxy`.

In all the solved benchmarks, we observed that it is better to perform branching only on non-auxiliary variables. Table 1 compares this branching (nb) with the normal branching (b) for the instances in [4]. The gains of the new branching scheme are clear; for example, we solve up to 3 times more instances of clique trees using the sequential direct encoding `s-dir`. For the warehouse instances, the new branching scheme reduces the time needed to solve one instance. In the rest of experiments we assume that the branching is performed only on non-auxiliary variables. In all the tables, the cutoff time is of 30 minutes.

The left plot of Figure 1 compares standard, regular, and sequential encodings of Random Max-CSP instances with the minimal support encoding `supc`. We display encoding `supc` because it is the best performing encoding for this benchmark. The instances were obtained with a generator of uniform random binary CSPs that implements the so-called model B: in the class $\langle n, d, p_1, p_2 \rangle$ with n variables of domain size d, we choose a random subset of exactly $p_1 n(n-1)/2$ constraints (rounded to the nearest integer), each with exactly $p_2 d^2$ conflicts (rounded to the nearest integer); p_1 may be thought of as the *density* of the problem and p_2 as the *tightness* of constraints. The difficulty of the instances depends on the selected values for n, d, p_1 and p_2. We selected values that allowed to solve the instances in a reasonable amount of time. We observe that

Table 2. Comparison between sequential encodings and regular encodings

		supc		supl		dir		supxy	
Kbtree (t)	#	sequential	regular	sequential	regular	sequential	regular	sequential	regular
10	50	1.24(50)	0.36(50)	0.10(50)	**0.07(50)**	15.01(50)	1.58(50)	89.18(44)	150.22(47)
20	50	57.07(50)	70.91(50)	**46.02(50)**	57.12(50)	345.30(49)	375.07(48)	0.00(0)	0.00(0)
30	50	**562.56(40)**	627.38(36)	559.65(38)	664.75(35)	1440.02(2)	0.00(0)	0.00(0)	0.00(0)
40	50	**1063.29(4)**	1341.48(2)	1175.17(3)	1714.93(2)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
Solved	450	**144**	138	141	137	101	98	44	47
		supc		supl		dir		supxy	
Warehouses	#	sequential	regular	sequential	regular	sequential	regular	sequential	regular
warehouse	2	2.36(1)	2.43(1)	2.38(1)	2.46(1)	2.35(1)	2.43(1)	1.34(1)	**1.44(1)**
Solved instances	2	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

the sequential encoding is up to one order of magnitude faster than the standard and regular encodings. The right plot compares the different sequential encodings (direct, minimal and support) defined in this paper. We observe that the minimal encoding is the best performing except for large values of p, where the support encoding dominates. For lower values of p, there is a big gap between the minimal and support encodings. It is also remarkable the superiority of the minimal encoding wrt the direct encoding.

Table 2 compares sequential encodings with regular encodings on the instances used in [4]. Standard encodings are not included because they are worse than regular encodings [4]. We see that, in general, the sequential encodings outperform the regular encodings on both the time needed to solve an instance and the number of solved instances. We also see that the minimal encodings are the best performing encodings.

Finally, we notice that our encodings may be easily extended with weigths because there is exactly one violated clause for every violated constraint, as well as that the direct encoding may incorporate non-binary constraints. As future work, we plan to investigate structural properties of encodings that may be useful to predict their performance.

References

1. Ansótegui, C., Manyà, F.: Mapping Problems with Finite-Domain Variables to Problems with Boolean Variables. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 1–15. Springer, Heidelberg (2005)
2. Argelich, J., Cabiscol, A., Lynce, I., Manyà, F.: Encoding Max-CSP into Partial Max-SAT. In: ISMVL 2008 (2008)
3. Argelich, J., Cabiscol, A., Lynce, I., Manyà, F.: Modelling Max-CSP as Partial Max-SAT. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 1–14. Springer, Heidelberg (2008)
4. Argelich, J., Cabiscol, A., Lynce, I., Manyà, F.: Regular encodings from Max-CSP into Partial Max-SAT. In: ISMVL 2009 (2009)
5. Argelich, J., Li, C.M., Manyà, F.: An improved exact solver for Partial Max-SAT. In: NCP-2007, pp. 230–231 (2007)
6. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)
7. Walsh, T.: SAT v CSP. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, pp. 441–456. Springer, Heidelberg (2000)

Cardinality Networks and Their Applications

Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and
Enric Rodríguez-Carbonell*

Abstract. We introduce *Cardinality Networks*, a new CNF encoding of cardinality constraints. It improves upon the previously existing encodings such as the sorting networks of [ES06] in that it requires much less clauses and auxiliary variables, while arc consistency is still preserved: e.g., for a constraint $x_1 + \ldots + x_n \leq k$, as soon as k variables among the x_i's become true, unit propagation sets all other x_i's to false. Our encoding also still admits *incremental strengthening:* this constraint for any smaller k is obtained without adding any new clauses, by setting a single variable to false.

Here we give precise recursive definitions of the clause sets that are needed and give detailed proofs of the required properties. We demonstrate the practical impact of this new encoding by careful experiments comparing it with previous encodings on real-world instances.

1 Introduction

Compared with other systematic constraint solving techniques, SAT solvers have many advantages for non-expert users as extremely efficient off-the-shelf black boxes that moreover require no tuning regarding variable (or value) selection heuristics. Therefore quite some work has been devoted to finding good propositional encodings for many kinds of constraints.

A particularly important class of constraints are the *cardinality constraints*, i.e., constraints of the form $x_1 + \ldots + x_n \ \# \ k$ where k is a natural number and $\# \ \in \{<, \leq, =, \geq, >\}$.

Cardinality constraints appear in many practical problem contexts, such as timetabling, scheduling, or pseudo-boolean constraint solving. For instance, given an input formula F over n variables $x_1, \ldots, x_n$, one may be interested in finding a model of F in which at most k variables are set to true. For this, one can add the clauses encoding the constraint $x_1 + \ldots + x_n \leq k$. Going beyond, for instance for the *min-ones* problem for F, that is, finding a model with the minimal number of true variables, one can *incrementally strengthen* the constraint for successively lower k until it becomes unsatisfiable. In fact, cardinality constraints frequently occur in other optimization problems too. For example, the Max-SAT problem consists of, given a set of clauses $S = \{C_1, \ldots, C_n\}$, finding an assignment A that satisfies the maximal number of clauses in S. One way of doing this is

* Technical Univ. of Catalonia, Barcelona. All authors partially supported by Spanish Min. of Educ. and Science through the LogicTools-2 project (TIN2007-68093-C02-01). The first author is also partially supported by FPI grant TIN2004-03382.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 167–180, 2009.

to add a fresh indicator variable x_i to each clause, getting $\{C_1 \vee x_1, \ldots, C_n \vee x_n\}$ and incrementally strengthening the constraint $x_1 + \ldots + x_n \leq k$. In general, it is typical to see situations where n is much larger than k.

This kind of applications of cardinality constraints has been very elegantly handled in MiniSAT and its extension to pseudo-boolean constraints [ES06]. There, one encoding for cardinality constraints is based on *sorting networks* with inputs $x_1, \ldots, x_n$ and output $y_1, \ldots, y_n$, such that if exactly k input variables are true, then $y_1, \ldots, y_k$ will become true and $y_{k+1}, \ldots, y_n$ will be false. For enforcing the constraint $x_1 + \ldots + x_n \leq k$, it then suffices to set y_{k+1} to false, and incrementally strengthening the constraint can be done by setting to false y_p's with successively smaller p.

In [ES06] it is also proved that for the CNF encoding of sorting networks unit propagation preserves arc consistency. For instance, for a constraint of the form $x_1 + \ldots + x_n \leq k$, as soon as k variables among the x_i's become true, unit propagation sets all other x_i's to false. The proof of arc consistency given in [ES06] relies on general properties of sorting networks.

Here we give recursive definitions for this kind of networks that, given sequences of input variables, return a sequence of output variables and a set of clauses. The required arc-consistency properties under unit propagation can be directly proved by induction from these definitions. Our starting point will be a deconstruction of the odd-even merge sorting networks of [Bat68], focussing on their specific use for encoding cardinality constraints in SAT.

For this purpose, and for allowing the reader to become familiar with the notations and methodology of this paper, in Section 3 we first define *Half Merging Networks* and *Half Sorting Networks*, which require only half as many clauses as their standard versions while preserving all desired properties.

As said, in many applications, it is typical to find cardinality constraints $x_1 + \ldots + x_n \ \# \ k$ where n is much larger than k. This motivated us to look for encodings that exploit this fact. In Section 4 we introduce *Cardinality Networks* which require $O(n \log^2 k)$ clauses instead of $O(n \log^2 n)$ as in previous approaches. In addition, Cardinality Networks also leverage the advantages from the use of Half Merging and Half Sorting Networks. All definitions, properties and proofs in this section and in Section 3 are for cardinality constraints of the form $x_1 + \ldots + x_n \leq k$. Therefore, in Section 5 we extend them to the other cases such as $\geq$ and $=$, and to *range constraints* of the form $k \leq x_1 + \ldots + x_n \leq k'$.

In Section 6 we demonstrate the practical impact of this new encoding by careful experiments comparing it with previous encodings on real-world instances and we conclude in Section 7. Because of space limitation, not all results are proved in the paper.

2 Preliminaries

Let P be a fixed finite set of propositional variables. If $p \in P$, then p and $\overline{p}$ are *literals* of P. The *negation* of a literal l, written $\overline{l}$, denotes $\overline{p}$ if l is p, and p if l is $\overline{p}$. A *clause* is a disjunction of literals $l_1 \vee \ldots \vee l_n$. A CNF *formula* is a conjunction

of one or more clauses $C_1 \wedge \ldots \wedge C_n$. When it leads to no ambiguities, we will sometimes consider such a formula as the set of its clauses.

A (partial truth) *assignment* M is a set of literals such that $\{p, \overline{p}\} \subseteq M$ for no p. A literal l is *true* in M if $l \in M$, is *false* in M if $\overline{l} \in M$, and is *undefined* in M otherwise. A clause C is true in M if at least one of its literals is true in M. A formula F is true in M if all its clauses are true in M. In that case, M is a *model* of F. The systems that decide whether a formula has a model or not are called *SAT solvers*.

Most state-of-the-art SAT solvers are based on extensions of the DPLL algorithm [DP60, DLL62]. The main inference rule in DPLL is known as *unit propagation*. Given a set of clauses S and an empty assignment M, clauses are sought in which all literals are false but one, say l, which is undefined (initially only clauses of size one satisfy this condition). This literal l is then added to M and the process is iterated until reaching a fix point. If U is the set of all literals that have been added to the assignment in this process, we will denote this fact by $S \models_{up} U$.

In this paper we will work with *cardinality constraints* $a_1 + \ldots + a_n \ \# \ k$, where $\# \ \in \{\leq, \geq, =\}$, the a_i's are propositional variables and k is a natural number. An assignment M *satisfies* such a constraint if at most ($\leq$), at least($\geq$) or exactly ($=$) k literals in $\{a_1, \ldots, a_n\}$ are true in M. The aim of this paper is, given a set of cardinality constraints C, to obtain a CNF formula S such that looking for assignments satisfying C is equivalent to looking for models of S. Moreover this S should be as small as possible and, whenever a concrete value for a variable in a constraint can be inferred, this should be detected by unit propagation on S.

In what follows, we consider *variable sequences*, or simply *sequences*, which are ordered lists of distinct propositional variables, written $\langle x_1 \ldots x_n \rangle$, and denoted by capital letters $A, B, C, \ldots$ Unless stated otherwise, these lists always have length $n = 2^m$, for some $m \geq 0$. When necessary these lists will be seen as sets, so that we can consider subsets of their variables.

Sometimes new *fresh* variables, that is, distinct new variables, will be introduced, These will always be denoted by the (possibly subscripted or primed) letters c, d, e.

3 Half Merging and Half Sorting Networks

In this section we introduce *Half Merging Networks* and *Half Sorting Networks*, which are like the *Sorting Networks* based on odd-even merges of [Bat68, ES06], but only need half of the clauses. The definitions and properties that are given will be used later on and allow the reader to become familiar with our notations and methodology. We remind that all the definitions in this section and in Section 4 are designed to be used in constraints of the form $x_1 + \ldots + x_n \leq k$, and that we implicitly assume that all sequences have size 2^m for some $m \geq 0$. In Section 5 we explain how to treat the case where n is not a power of two.

3.1 Half Merging Networks

Given two sequences A and B of length n, the *Half Merging Network of A and B*, denoted $HMerge(A, B)$, is a pair (C, S), where C is a sequence of length $2n$ and S is a set of clauses, defined as follows.

For sequences of length 1 we define:

$$HMerge(\ \langle a\rangle,\ \langle b\rangle\ )\ =\ (\ \langle\ c_1\ c_2\ \rangle,\ \{\ \overline{a}\vee\overline{b}\vee c_2,\ \ \overline{a}\vee c_1,\ \ \overline{b}\vee c_1\ \ \}\ )$$

For sequences of length $n > 1$ we define:

$$HMerge(\ \langle a_1 \dots a_n\rangle,\ \langle b_1 \dots b_n\rangle\ )\ =\ (\ \langle\ d_1\ c_2\ \dots\ c_{2n-1}\ e_n\ \rangle,\ \ S_{odd} \cup S_{even} \cup S'\ )$$

recursively in terms of the odd and the even subsequences:

$$HMerge(\ \langle a_1\ a_3 \dots a_{n-1}\rangle,\ \langle b_1\ b_3 \dots b_{n-1}\rangle\ )\ =\ (\ \langle d_1\ \dots\ d_n\rangle,\ S_{odd}\ \ ),$$
$$HMerge(\ \langle a_2\ a_4 \dots a_n\ \ \ \rangle,\ \langle b_2\ b_4 \dots b_n\ \ \ \rangle\ )\ =\ (\ \langle e_1\ \dots\ e_n\rangle,\ S_{even}\ ),$$

where the clause set S' is: $\bigcup_{i=1}^{n-1}\{\ \overline{d}_{i+1}\vee\overline{e}_i\vee c_{2i+1},\ \ \overline{d}_{i+1}\vee c_{2i},\ \ \overline{e}_i\vee c_{2i}\ \}$.

Example 1. Intuitively, a (Half) Merging Network merges two sequences of input variables $\langle a_1 \dots a_n\rangle$ and $\langle b_1 \dots b_n\rangle$ that are already sorted into a single sorted output sequence $\langle c_1 \dots c_{2n}\rangle$, and the required unit propagation is that if $a_1 \dots a_p$ and $b_1 \dots b_q$ are true, then the first $p + q$ output variables will become true (Lemma 1 below), and (roughly speaking) if in addition c_{p+q+1} is set to false, then also a_{p+1} and b_{q+1} will become false (Lemma 2).

Let us take $HMerge(\ \langle a_1\ a_2\rangle,\ \langle b_1\ b_2\rangle\ )$, which is $(\ \langle d_1\ c_2\ c_3\ e_2\rangle, S)$ with S being the set of clauses:

$$\begin{array}{lll}
\overline{a}_1\vee\overline{b}_1\vee d_2 & \overline{a}_2\vee\overline{b}_2\vee e_2 & \overline{d}_2\vee\overline{e}_1\vee c_3\\
\overline{a}_1\vee d_1 & \overline{a}_2\vee e_1 & \overline{d}_2\vee c_1\\
\overline{b}_1\vee d_1 & \overline{b}_2\vee e_1 & \overline{e}_1\vee c_2
\end{array}$$

The partial assignments $(a_1, a_2) = (1, 0)$ and $(b_1, b_2) = (0, 0)$ cause S to unit propagate the first output (d_1), but not the second one (c_2). If we add another 1 to the input, for example $(a_1, a_2) = (1, 1)$, then both d_1 and c_2 get propagated, but not c_3. For propagating c_3 we need to add another input 1, e.g, setting $(b_1, b_2) = (1, 0)$, but $(b_1, b_2) = (0, 1)$ would not do it, since this propagation only works if all ones appear as a prefix in the input sequences, which will always be the case in our uses of *HMerge*. With inputs $(a_1, a_2) = (1, 0)$ and $(b_1, b_2) = (1, 0)$, and setting c_3 to false, unit propagation will set a_2 and b_2 to false. Similar properties about propagation of ones and zeros will hold in all the constructions in this paper and will be precisely stated in each case. □

Lemma 1. *If* $HMerge(\ \langle a_1 \dots a_n\rangle,\ \langle b_1 \dots b_n\rangle\ )\ =\ (\ \langle c_1\ \dots\ c_{2n}\rangle,\ \ S\ )$ *and* $p, q \in \mathbb{N}$ *with* $1 \le p, q \le n$, *then* $S \cup \{a_1\ \dots\ a_p\ b_1 \dots b_q\} \models_{up}\ c_1, \dots, c_{p+q}$.

Lemma 2. *Let* $HMerge(\ \langle a_1 \dots a_n\rangle,\ \langle b_1 \dots b_n\rangle\ )$ *be* $(\ \langle c_1\ \dots\ c_{2n}\rangle,\ \ S\ )$, *and* $p, q \in \mathbb{N}$ *with* $p, q \le n$.

If $p < n$ *and* $q < n$ *then* $S \cup \{a_1, \dots, a_p, b_1, \dots, b_q, \overline{c}_{p+q+1}\} \models_{up}\ \overline{a}_{p+1},\ \overline{b}_{q+1}$.
If $p = n$ *and* $q < n$ *then* $S \cup \{a_1, \dots, a_p, b_1, \dots, b_q, \overline{c}_{p+q+1}\} \models_{up}\ \overline{b}_{q+1}$.
If $p < n$ *and* $q = n$ *then* $S \cup \{a_1, \dots, a_p, b_1, \dots, b_q, \overline{c}_{p+q+1}\} \models_{up}\ \overline{a}_{p+1}$.

Lemma 3. *Given A and B sequences of length n, the Half Merging Network $HMerge(A, B)$ contains $O(n \log n)$ clauses with $O(n \log n)$ auxiliary variables.*

3.2 Half Sorting Networks

Given a sequence A of length $2n$, the *Half Sorting Network of A*, denoted $HSort(A)$, is a pair (C, S), where C is a sequence of length $2n$ and S is a set of clauses, defined as follows.

For sequences of length 2 we define:

$$HSort(\ \langle a\ b\rangle\ )\ =\ HMerge(\ \langle a\rangle,\ \langle b\rangle\ )$$

For sequences of length $2n > 2$ we define:

$$HSort(\ \langle a_1\ \ldots\ a_{2n}\rangle\ )\ =\ (\ \langle c_1\ \ldots\ c_{2n}\rangle,\ \ S_D \cup S_{D'} \cup S_M\ )$$

recursively in terms of two subsequences of size n:

$$HSort(\ \langle\ a_1\ \ \ldots\ \ a_n\ \rangle\ )\ =\ (\ \langle d_1\ \ldots\ d_n\rangle,\ S_D\ ),$$
$$HSort(\ \langle\ a_{n+1} \ldots a_{2n}\rangle\ )\ =\ (\ \langle d'_1\ \ldots\ d'_n\rangle,\ S_{D'}\ ),$$

and the merge of them

$$HMerge(\ \langle d_1 \ldots d_n\ \rangle,\ \langle d'_1 \ldots d'_n\ \rangle\ )\ =\ (\ \langle c_1\ \ldots\ c_{2n}\rangle,\ S_M\ ),$$

Lemma 4. *Given a sequence A of length n, the Half Sorting Network $HSort(A)$ contains $O(n \log^2 n)$ clauses with $O(n \log^2 n)$ auxiliary variables.*

Similar properties to the ones of Half Merging Networks also hold here, but without the requirement that the input ones are at prefixes: (i) if *any* p input variables are set to true, the first p output variables are unit propagated (Lemma 5), and (ii) if in addition the $p+1$-th output is set to false, the remaining input variables are set to false (Lemma 6), hence not allowing more than p input variables to be true.

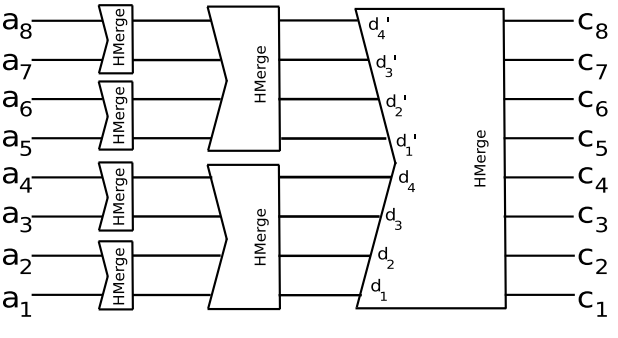


Fig. 1. *HSort* with input $\langle a_1 \ldots a_8\rangle$ and output $\langle c_1 \ldots c_8\rangle$

Lemma 5. *Let HSort*(A) *be* $(\langle c_1 \dots c_{2n} \rangle, S)$ *and let* $A' \subseteq A$ *with* $|A'| = p$. *Then,*

$$S \cup A' \models_{up} c_1, \dots, c_p$$

Lemma 6. *Let HSort*(A) *be* $(\langle c_1 \dots c_{2n} \rangle, S)$ *and let* $A' \subsetneq A$ *with* $|A'| = p < 2n$. *Then,*

$$S \cup A' \cup \overline{c}_{p+1} \models_{up} \overline{a}_j \text{ for all } a_j \in (A - A')$$

4 Cardinality Networks

Here we exploit the fact that in cardinality constraints $x_1 + \dots + x_n \leq k$ it is frequently the case that n is much larger than k. We introduce *Cardinality Networks* which require $O(n \log^2 k)$ clauses instead of $O(n \log^2 n)$ as in previous approaches. A main ingredient for Cardinality Networks are the *Simplified Merging Networks*, which we introduce first.

4.1 Simplified Merging Networks

If we are only interested in the (maximal) $n+1$ bits of the output (instead of the $2n$ original ones), Half Merging Networks can be further simplified. Given two sequences A and B of length n, the *Simplified Merging Network of A and B*, denoted $SMerge(A, B)$, is a pair (C, S), where C is a sequence of length $n+1$ and S is a set of clauses, defined as follows. For $n = 1$, we have

$$SMerge(\langle a \rangle, \langle b \rangle) = (\langle c_1, c_2 \rangle, \{\overline{a} \vee \overline{b} \vee c_2, \overline{a} \vee c_1, \overline{b} \vee c_1\})$$

The case $n > 1$ is defined

$$SMerge(\langle a_1 \dots a_n \rangle, \langle b_1 \dots b_n \rangle) = (\langle d_1\, c_2 \dots c_{n+1} \rangle, S_{odd} \cup S_{even} \cup S')$$

recursively in terms of the odd and the even subsequences,

$$SMerge(\langle a_1\, a_3 \dots a_{n-1} \rangle, \langle b_1\, b_3 \dots b_{n-1} \rangle) = (\langle d_1 \dots d_{\frac{n}{2}+1} \rangle, S_{odd})$$
$$SMerge(\langle a_2\, a_4 \dots a_n \rangle, \langle b_2\, b_4 \dots b_n \rangle) = (\langle e_1 \dots e_{\frac{n}{2}+1} \rangle, S_{even})$$

where the clause set S' is:

$\bigcup_{i=1}^{\frac{n}{2}} \{\overline{d}_{i+1} \vee \overline{e}_i \vee c_{2i+1},\ \overline{d}_{i+1} \vee c_{2i},\ \overline{e}_i \vee c_{2i}\}$.

Remark. We have defined Simplified Merging Networks with $n+1$ outputs because this $n+1$-th bit is needed for the odd recursive case: $d_{\frac{n}{2}+1}$ is used in the clause set S'. But output $e_{\frac{n}{2}+1}$ from the even subcase is not used, and the $n+1$-th bit is not used either in the Cardinality Networks defined below. This fact can be exploited for a slightly further optimization in our encodings by using Simplified Merging Networks with n outputs for these subcases, but for clarity of explanation we have chosen not to do so here.

We now precisely state the propagation properties of Simplified Merging Networks. Lemma 7 is the equivalent of Lemma 1, proving that $p+q$ inputs ones properly placed (e.g. as prefixes in the input sequences), unit propagate the first $p+q$ outputs. After that, Lemma 8, the equivalent of Lemma 2, proves how zeros can be propagated from outputs to inputs.

Lemma 7. *If* $SMerge(\ \langle a_1 \dots a_n\rangle,\ \langle b_1 \dots b_n\rangle\ )\ =\ (\ \langle c_1\ \dots\ c_{n+1}\rangle,\ S\ )$ *and* $p, q \in \mathbb{N}$ *with* $1 \le p+q \le n+1$, *then* $S \cup \{a_1, \dots, a_p, b_1, \dots, b_q\} \models_{up} c_{p+q}$.

Proof. (By induction on n). If $n = 1$, we have

$$SMerge(\ \langle a\rangle, \langle b\rangle)\ =\ (\ \langle c_1, c_2\rangle,\ \{\ \overline{a} \vee c_1, \overline{b} \vee c_1, \overline{a} \vee \overline{b} \vee c_2\ \}\ ).$$

If $p = 0$, $q = 1$ then setting b clearly propagates c_1. Similarly, if $p = 1$, $q = 0$, setting a propagates c_1. Otherwise, $p = 1$, $q = 1$, and a and b propagate c_2.

For the induction step ($n > 1$) we consider four different cases, depending on whether p and q are odd or even:

<u>*CASE 1:*</u> p is odd and q even. (Let $p = 2p' + 1$ and $q = 2q'$).
Let us focus on the odd part of *SMerge*:

$$SMerge(\ \langle a_1\, a_3 \dots a_{n-1}\rangle,\ \langle b_1\, b_3 \dots b_{n-1}\rangle\ )\ =\ (\ \langle d_1\ \dots\ d_{\frac{n}{2}+1}\rangle,\ S_{odd}\ ).$$

In $\langle a_1\, a_2 \dots a_p\rangle$ there are $p'+1$ odd indices, namely $\{1, 3, \dots, 2p'+1\}$. Similarly, in $\langle b_1\, b_2 \dots b_q\rangle$ there are q' odd indices, namely $\{1, 3, \dots, 2q'-1\}$. Hence, by IH we have $S_{odd} \cup \{a_1, \dots, a_{2p'+1}, b_1, \dots, b_{2q'-1}, \} \models_{up} d_{p'+q'+1}$ (note that $1 \le (p'+1) + q' \le \frac{n}{2}+1$).

Now, let us take the even part of *SMerge*:

$$SMerge(\ \langle a_2\, a_4 \dots a_n\rangle,\ \langle b_2\, b_4 \dots b_n\rangle\ )\ =\ (\ \langle e_1\ \dots\ e_{\frac{n}{2}+1}\rangle,\ S_{even}\ )$$

In $\langle a_2\, a_4 \dots a_p\rangle$ there are p' even indices, namely $\{2, 4, \dots, 2p'\}$. Similarly, in $\langle b_2\, b_4 \dots b_q\rangle$ there are q' even indices, namely $\{2, 4, \dots, 2q'\}$. Hence, by IH we have $S_{even} \cup \{a_2, \dots, a_{2p'}, b_2, \dots, b_{2q'}, \} \models_{up} e_{p'+q'}$ (note that $1 \le p' + q' \le \frac{n}{2}+1$).

Finally, since $1 \le p' + q' \le \frac{n}{2}$ the clause $\overline{d}_{p'+q'+1} \vee \overline{e}_{p'+q'} \vee c_{2p'+2q'+1}$ belongs to S, and hence literal $c_{2p'+2q'+1}$ can be unit propagated, as we wanted to prove.

<u>*CASE 2:*</u> p is even and q odd. (Symmetric to the previous one).

<u>*CASE 3:*</u> p and q are odd. (Let $p = 2p' + 1$ and $q = 2q' + 1$).
We will now use only the odd part of *SMerge*:

$$SMerge(\ \langle a_1\, a_3 \dots a_{n-1}\rangle,\ \langle b_1\, b_3 \dots b_{n-1}\rangle\ )\ =\ (\ \langle d_1\ \dots\ d_{\frac{n}{2}+1}\rangle,\ S_{odd}\ ).$$

In $\langle a_1\, a_2 \dots a_p\rangle$ there are $p'+1$ odd indices, namely $\{1, 3, \dots, 2p'+1\}$. Similarly, in $\langle b_1\, b_2 \dots b_q\rangle$ there are $q'+1$ odd indices, namely $\{1, 3, \dots, 2q'+1\}$. Hence, by IH we have $S_{odd} \cup \{a_1, \dots, a_{2p'+1}, b_1, \dots, b_{2q'+1}, \} \models_{up} d_{p'+q'+2}$ (note that, using that n is even, one can see that $1 \le (p'+1) + (q'+1) \le \frac{n}{2}+1$).

Now, since $1 \le p' + q' + 1 \le \frac{n}{2}$, the clause $\overline{d}_{p'+q'+2} \vee c_{2p'+2q'+2}$ belongs to S, the literal $c_{2p'+2q'+2}$ can be unit propagated.

<u>*CASE 4:*</u> p and q are even. (Let $p = 2p'$ and $q = 2q'$).
We will now only use the even part of *SMerge*:

$$SMerge(\ \langle a_2\, a_4 \dots a_n\rangle,\ \langle b_2\, b_4 \dots b_n\rangle\ )\ =\ (\ \langle e_1\ \dots\ e_{\frac{n}{2}+1}\rangle,\ S_{even}\ )$$

In $\langle a_2\, a_4 \ldots a_p \rangle$ there are p' even indices, namely $\{2, 4, \ldots, 2p'\}$. Similarly, in $\langle b_2\, b_4 \ldots b_q \rangle$ there are q' even indices, namely $\{2, 4, \ldots, 2q'\}$. Hence, by IH we have $S_{even} \cup \{a_2, \ldots, a_{2p'}, b_2, \ldots, b_{2q'}, \} \models_{up} e_{p'+q'}$ (note that $1 \leq p' + q' \leq \frac{n}{2} + 1$).

Now, using that n is even, one can see that $1 \leq p'+q' \leq \frac{n}{2}$ and hence the clause $\overline{e}_{p'+q'} \vee c_{2p'+2q'}$ belongs to S, allowing one to propagate the literal $c_{2p'+2q'}$. □

Lemma 8. *Let SMerge*$(\ \langle a_1 \ldots a_n \rangle,\ \langle b_1 \ldots b_n \rangle\ )$ *be* $(\ \langle\, c_1\ \ldots\ c_{n+1}\, \rangle,\ \ S\ )$, *and* $p, q \in \mathbb{N}$ *with* $p + q \leq n$.

If $p < n$ *and* $q < n$ *then* $S \cup \{a_1, \ldots, a_p, b_1, \ldots, b_q, \overline{c}_{p+q+1}\} \models_{up} \overline{a}_{p+1}, \overline{b}_{q+1}$.
If $p = n$ *and* $q = 0$ *then* $S \cup \{a_1, \ldots, a_n, \overline{c}_{n+1}\} \models_{up} \overline{b}_1$.
If $p = 0$ *and* $q = n$ *then* $S \cup \{b_1, \ldots, b_n, \overline{c}_{n+1}\} \models_{up} \overline{a}_1$.

Lemma 9. *Given A and B sequences of length n, the Simplified Merging Network* $SMerge(A, B)$ *contains* $O(n \log n)$ *clauses with* $O(n \log n)$ *auxiliary variables.*

4.2 K-Cardinality Networks

Given a sequence A of length $n = m \times k$ with $k = 2^r$ and $m \in \mathbb{N}$, the *k-Cardinality Network of A*, denoted $Card(A, k)$, is a pair (C, S), where C is a sequence of length k and S is a set of clauses, defined as follows.

For sequences of length k, we define:

$$Card(\ \langle a_1\ \ldots\ a_k \rangle,\ k\ )\ =\ HSort(\ \langle a_1\ \ldots\ a_k \rangle\ )$$

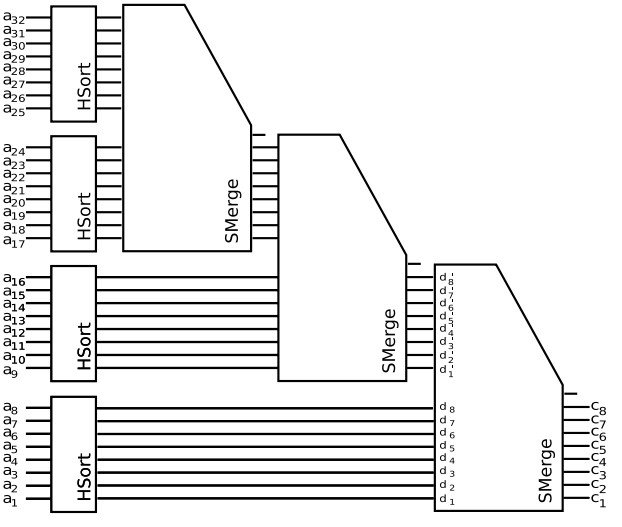


Fig. 2. Representation of $Card(\langle a_1 \ldots a_{32} \rangle, 8)$ with output $\langle c_1 \ldots c_8 \rangle$

For sequences of length $n > k$ we define:

$$Card(\langle a_1 \ldots a_n \rangle, k) = (\langle c_1 \ldots c_k \rangle, \; S_D \cup S_{D'} \cup S_M)$$

recursively in terms of subsequences of sizes k and $n-k$:

$$Card(\langle a_1 \ldots a_k \rangle, k) = (\langle d_1 \ldots d_k \rangle, S_D),$$
$$Card(\langle a_{k+1} \ldots a_n \rangle, k) = (\langle d'_1 \ldots d'_k \rangle, S_{D'}),$$

and a simplified merge of them (note that its last output is not used)

$$SMerge(\langle d_1 \ldots d_k \rangle, \langle d'_1 \ldots d'_k \rangle) = (\langle c_1 \ldots c_{k+1} \rangle, S_M),$$

Lemma 10. *Given a sequence A of length $n = m \times k$, the k-Cardinality Network $Card(A, k)$ contains $O(n \log^2 k)$ clauses with $O(n \log^2 k)$ auxiliary variables.*

Again, the usual properties of how zeros and ones are unit propagated follow. Their proofs are analogous to the ones of Lemma 5 and Lemma 6.

Lemma 11. *If $Card(A, k) = (\langle c_1 \ldots c_k \rangle, S)$ and $A' \subseteq A$ with $|A'| = p \leq k$, then*

$$S \cup A' \models_{up} c_1, \ldots, c_p$$

Proof. Sequence A will be of the form $\langle a_1 \ldots a_n \rangle$ with $n = m \times k$. We will prove the lemma by induction on m.

If $m = 1$, we have $Card(\langle a_1, \ldots a_k \rangle, k) = HSort(\langle a_1, \ldots, a_k \rangle)$. Using lemma 5 we conclude that $\{c_1, \ldots, c_p\}$ are unit propagated.

For the induction step ($m > 1$) we have:

$$\begin{aligned}
Card(\langle a_1 \ldots a_n \rangle, k) &= (\langle c_1 \ldots c_k \rangle, \; S_D \cup S_{D'} \cup S_M), \text{ with} \\
Card(\langle a_1 \ldots a_k \rangle, k) &= (\langle d_1 \ldots d_k \rangle, \; S_D), \\
Card(\langle a_{k+1} \ldots a_n \rangle, k) &= (\langle d'_1 \ldots d'_k \rangle, \; S_{D'}) \text{ and} \\
SMerge(\langle d_1 \ldots d_k \rangle, \langle d'_1 \ldots d'_k \rangle,) &= (\langle c_1 \ldots c_{k+1} \rangle, \; S_M)
\end{aligned}$$

If we now consider the sets $A_D = A' \cap \{a_1, \ldots, a_k\}$, with size $|A_D| = p_D$, and $A_{D'} = A' \cap \{a_{k+1}, \ldots, a_n\}$, with $|A_{D'}| = p_{D'}$, by IH we have $A_D \cup S_D \models_{up} d_1, \ldots, d_{p_D}$ and $A_{D'} \cup S_{D'} \models_{up} d'_1, \ldots, d'_{p_{D'}}$. Now, by lemma 7 we know that $S_M \cup \{d_1, \ldots, d_{p_D}, d'_1, \ldots, d'_{p_{D'}}\} \models_{up} c_1, \ldots, c_{p_D + p_{D'}}$, which, since $p = p_D + p_{D'}$, concludes the proof. □

Theorem 1. *If $Card(\langle a_1 \ldots a_n \rangle, k) = (\langle c_1 \ldots c_k \rangle, S)$ and $A' \subsetneq A$ with size $|A'| = p < k$, then*

$$S \cup A' \cup \overline{c}_{p+1} \models_{up} \overline{a}_j \text{ for all } a_j \in (A \setminus A')$$

Proof. We have that $n = m \times k$, and we will prove the lemma by induction on m.

If $m = 1$, we have $Card(\langle a_1, \ldots a_k \rangle, k) = HSort(\langle a_1, \ldots, a_k \rangle)$ and in this case the theorem amounts to Lemma 6.

For the induction step ($m > 1$) we have:

$$
\begin{array}{lll}
\mathit{Card}(\ \langle\, a_1 \,\ldots\, a_n \,\rangle\ , k\ ) & = (\langle c_1 \ldots c_k \,\rangle,\ S_D \cup S_{D'} \cup S_M\ ), \text{ with} \\
\quad \mathit{Card}(\ \langle\, a_1 \,\ldots\, a_k \,\rangle,\ k\ ) & = (\ \langle d_1 \,\ldots\, d_k \rangle,\ S_D\ ), \\
\quad \mathit{Card}(\ \langle a_{k+1} \ldots a_n \rangle,\ k\ ) & = (\ \langle d'_1 \,\ldots\, d'_k \rangle,\ S_{D'}\ ) \text{ and} \\
\quad \mathit{SMerge}(\ \langle d_1 \ldots d_k \,\rangle,\ \langle d'_1 \ldots d'_k \,\rangle,\ ) & = (\ \langle c_1 \,\ldots\, c_{k+1} \rangle,\ S_M\ )
\end{array}
$$

If we now consider the sets $A_D = A' \cap \{a_1, \ldots, a_k\}$, with size $|A_D| = p_D$, and $A_{D'} = A' \cap \{a_{k+1}, \ldots, a_n\}$, with $|A_{D'}| = p_{D'}$, by Lemma 11 we know that $A_D \cup S_D \models_{up} \ d_1, \ldots, d_{p_D}$ and $A_{D'} \cup S_{D'} \models_{up} \ d'_1, \ldots, d'_{p_{D'}}$. Due to these propagated literals and knowing that $p = p_D + p_{D'} \leq k$ and both $p_D < k$ and $p_{D'} < k$, we obtain $S_M \cup \{d_1, \ldots, d_{p_D}, d'_1, \ldots, d'_{p_{D'}}, \overline{c}_{p+1}\} \models_{up} \ \overline{d}_{p_D+1}, \overline{d}'_{p_{D'}+1}$ by applying Lemma 8.

Finally these two unit propagations allow us to use the IH to infer that $S_D \cup A_D \cup \overline{d}_{p_D+1} \models_{up} \ \overline{a}_j$ for all $a_j \in (\{a_1 \ldots a_k\} - A_D)$ and also that $S_{D'} \cup A_{D'} \cup \overline{d}'_{p_{D'}+1} \models_{up} \ \overline{a}_j$ for all $a_j \in (\{a_{k+1} \ldots a_n\} - A_{D'})$, which concludes the proof. □

5 Application to SAT Solving and Extensions

In this section we show how to apply the previous constructions in practice and we further present some extensions:

• **Use of *Card* in practice.** Theorem 1 indicates how to apply the construction *Card* in practice. Assume we are given a formula F to which we want to impose the cardinality constraint $a_1 + \ldots a_n \leq p$. We should first find k, the smallest power of two with $k > p$ and consider the construction $\mathit{Card}(\langle a_1 \ldots a_{n+m} \rangle, k) = (\ \langle c_1, \ldots c_k \rangle, S\ )$. Note that we may need to add m extra variables to the input sequence to obtain a sequence of size multiple of k, but these variables are initially set to false and do not enlarge the search space. Now, the problem amounts to check the satisfiability of $F \wedge S \wedge \overline{c}_{p+1}$ since, due to Theorem 1, as soon as p variables in $\langle a_1, \ldots, a_{n+m} \rangle$ are set to true, the remaining ones will be unit propagated to false, hence disallowing any model not satisfying the cardinality constraint.

• **Incremental strengthening.** Another important feature of these encodings can be exploited in applications where one needs to solve a sequence of problems that only differ in that a cardinality constraint $a_1 + \ldots + a_n \leq p$ becomes increasingly stronger by decreasing p to p', as it happens in optimization problems. In this setting, we only need to assert the corresponding literal $\overline{c}_{p'+1}$, and the search can be resumed keeping all lemmas generated in the previous problems. Most state-of-the-art SAT solvers used as black boxes provide a user interface for doing this.

• **Constraints of the form $a_1 + \ldots a_n \geq p$.** For these type of constraints, we should first find k, the smallest power of two with $k \geq p$. After that, we should consider a new construction $\mathit{Card}^{\geq}(\langle a_1, \ldots, a_{n+m} \rangle, k) = (\langle c_1, \ldots, c_k \rangle, S)$, identical to $\mathit{Card}(A, k)$, except that its blocks *HMerge* and *SMerge* contain, in their basic case, the clauses $\{a \vee b \vee \overline{c}_1,\ \ a \vee \overline{c}_2,\ \ b \vee \overline{c}_2\}$ and, for the recursive case, the clause set S' is built from the clauses $\{d_{i+1} \vee \overline{c}_{2i+1},\ \ e_i \vee \overline{c}_{2i+1},\ \ d_{i+1} \vee e_i \vee \overline{c}_{2i}\}$. We have the following result:

Theorem 2. *If* $Card^{\geq}(\langle a_1 \dots a_n \rangle, k) = (\langle c_1 \dots c_k \rangle, S)$ *and* $A' \subsetneq A$ *with* $|A'| = n - p$*, for some* $p \in \mathbb{N}$ *with* $1 \leq p \leq k$*, then*

$$S \cup \overline{A'} \cup c_p \models_{up} a_j \text{ for all } a_j \in (A \setminus A'),$$

where $\overline{A'}$ *contains the negation of all variables of* A'*.*

This theorem ensures that, if we set c_p to true, as soon as $n - p$ literals are set to false, the remaining p will be set to true, hence forcing the constraint to be satisfied.

• **Constraints of the form** $p \leq a_1 + \dots a_n \leq q$**.** For these constraints, of which equality constraints $a_1 + \dots + a_n = p$ are a particular case, we should first find k, the smallest power of two such that $k > q$. Then, we will use another construction $Card^{rng}(\langle a_1, \dots, a_{n+m} \rangle, k) = (\langle c_1, \dots, c_k \rangle, S)$, identical to $Card(A, k)$, except that its blocks *HMerge* and *SMerge* contain, in their basic and recursive cases, all 6 mentioned clauses (the ones for *Card* and the ones for $Card^{\geq}$). This allows one to avoid encoding the two constraints independently, which would roughly duplicate the number of variables. For this construction, we have:

Theorem 3. *Let* $Card^{rng}(\langle a_1 \dots a_n \rangle, k) = (\langle c_1 \dots c_k \rangle, S)$ *and* $A' \subsetneq A$*.*

– *If* $|A'| = n - p$ *for some* $p \in \mathbb{N}$ *with* $1 \leq p \leq k$ *then*

$$S \cup \overline{A'} \cup c_p \models_{up} a_j \text{ for all } a_j \in (A \setminus A'),$$

– *If* $|A'| = p$ *for some* $p < k$ *then*

$$S \cup A' \cup \overline{c}_{p+1} \models_{up} \overline{a}_j \text{ for all } a_j \in (A \setminus A')$$

This theorem ensures that, if we set c_p and $\overline{c}_{q+1}$, then (i) as soon as $n - p$ variables are set to false, the remaining ones will be set to true and (ii) as soon as q variables are set to true, the remaining ones will be set to false, which forces the constraint to be satisfied.

• **Constraints** $a_1 + \dots + a_n \leq p$ **with** $p > \frac{n}{2}$. Note that Cardinality Networks were designed to improve upon Sorting Networks when n is much larger than p. If $p > \frac{n}{2}$ we can use the fact that the constraint above can be rewritten as $(1 - a_1) + \dots + (1 - a_n) \geq n - p$. The latter constraint, where now $n - p < \frac{n}{2}$, can be encoded using Cardinality Networks by simply changing the input variables by their negations.

6 Evaluation

We first show some statistics, for a constraint $a_1 + \dots + a_n \leq k$, about the number of variables and clauses[1] in Cardinality Networks compared with the

[1] Since for every ternary clause there are two binary clauses, the number of literals in the encodings is $\frac{7}{3}$ times the number of clauses.

Sorting Networks of [Bat68, ES06] (figures for our Half Sorting Networks are as for Sorting Networks, except that the number of clauses is halved). Cardinality Networks provide a huge advantage for small values of k, whereas for $k = \frac{n}{2}$ (its worst case) there is still more than a factor-two advantage due to the use of Half Sorting/Merging Networks instead of full ones.

	Sorting Network		Cardinality Network							
			k=5		k=10		k=15		k=n/2	
n	vars	clauses	vars	clauses	vars	clauses	vars	clauses	vars	clauses
10^5	$18 \cdot 10^6$	$54 \cdot 10^6$	$77 \cdot 10^4$	$12 \cdot 10^5$	$12 \cdot 10^5$	$18 \cdot 10^5$	$12 \cdot 10^5$	$19 \cdot 10^5$	$15 \cdot 10^6$	$23 \cdot 10^6$
10^4	$15 \cdot 10^5$	$45 \cdot 10^5$	$77 \cdot 10^3$	$12 \cdot 10^4$	$12 \cdot 10^4$	$18 \cdot 10^4$	$12 \cdot 10^4$	$19 \cdot 10^4$	$12 \cdot 10^5$	$19 \cdot 10^5$
10^3	48150	144403	7713	12065	12223	18825	12857	19771	39919	59879
10^2	2970	8855	773	1205	1251	1917	1325	2023	2279	3419

We now also assess the practical performance of the encodings. To the best of our knowledge there is no standard library for SAT benchmarks with cardinality constraints. However, there exists a very large and diverse source of realistic instances, namely the ones produced by the *msu4* algorithm [MSP08] where Max-SAT problems are reduced to a series of SAT problems with cardinality constraints.

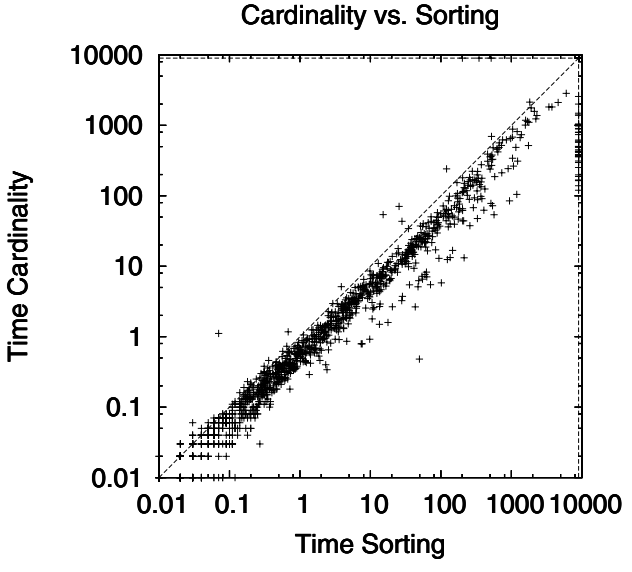


Fig. 3. Times in seconds and logarithmic scale is used

We have made a simple *msu4* implementation which, every time a non-trivial cardinality constraint is used (that is, that cannot be converted into a single clause or a set of unit literals), also writes the SAT + cardinality constraints problem into a file. We have run this prototype on all benchmarks used in the

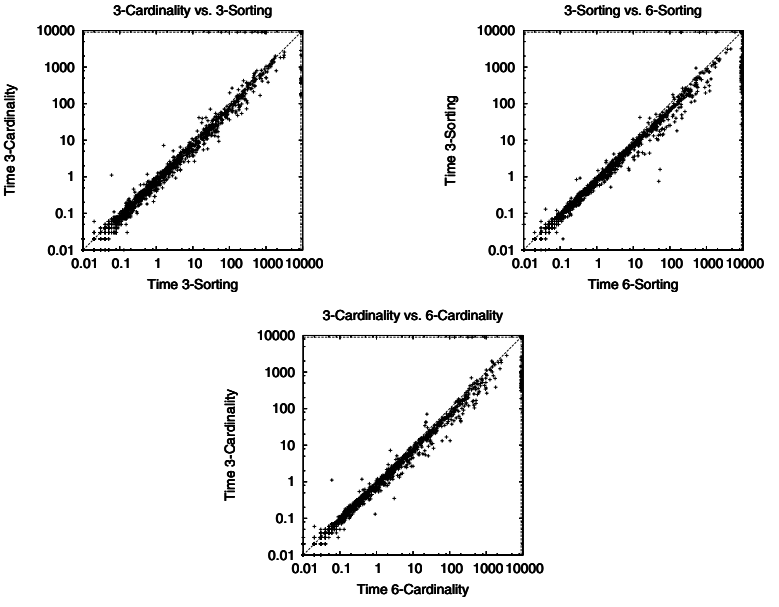


Fig. 4. Times in seconds and logarithmic scale is used. Best settings on the y-axes.

Partial Max-SAT division of the Third Max-SAT evaluation[2]. Hence, for every benchmark in this division (some 1800), we have created a *family* of SAT + cardinality constraints problems, usually between 2 and 10, which we believe constitute a large and diverse enough set of benchmarks. We run each one of them with Sorting and with Cardinality Networks on a 2Ghz Linux Quad-Core AMD using our Barcelogic SAT Solver that ranked 3rd in the 2008 SAT-Race[3]. Results are plotted in Figure 3, which shows a clear win for Cardinality Networks. Each cross represents the time to solve a family of benchmarks. Each benchmark was given 600 seconds and timing out in a single benchmark is counted as a timeout for the whole family (in the plot, these are the crosses in the vertical or horizontal lines).

One may wonder where the improvements come from the use of Half Merging/Sorting Networks (3 clauses instead of 6) or from the asymptotically smaller Cardinality Networks ($O(n \log^2 k)$ clauses and auxiliary variables vs. $O(n \log^2 n)$). The answer is: both, as one can see from Figure 4, where we also compare with **3-Sorting:** Half Sorting Networks as described in Section 3.2, and **6-Cardinality:** Cardinality Networks with *HMerge* and *SMerge* using all 6 clauses instead of only the 3 mentioned in Section 3.1 and Section 4.1. In particular, using 3 clauses has beneficial effects for both Sorting Networks and Cardinality Constraints.

[2] See `http://www.maxsat.udl.cat/08/index.php?disp=submitted-benchmarks`
[3] See `http://baldur.iti.uka.de/sat-race-2008/`

7 Conclusions and Further Work

SAT solvers can be used off the shelf, giving high performance push-button tools, i.e., tools that require no tuning for variable or value selection heuristics. In order to exploit these features optimally, it is important to develop a catalogue of encodings for the most common general-purpose constraints, in such a way that the SAT solver's unit propagation can efficiently preserve arc consistency.

The cardinality constraints we have studied here are certainly among the most ubiquitous ones. Therefore, apart from the aforementioned work [ES06], they have also been studied elsewhere. For instance in [Sin05] two encodings are given, one requiring $7n$ clauses and $2n$ auxiliary variables, and another one based on n unary $k - bit$ counters c_i denoting the number of true inputs among $x_1 \dots x_i$; this latter encoding preserves arc consistency like ours, but it requires $O(n \cdot k)$ clauses and new variables; in [BB03] arc consistency is also preserved but $O(n^2)$ clauses and $O(n \log n)$ variables are required. In [SL07] the case of $k = 1$ is studied, showing how a state-of-the-art SAT solver can be adapted to diminish the noise introduced by the auxiliary variables.

Our approach is based on precise (recursive) definitions of the generated clause sets and on inductive proofs for the arc consistency properties, combined with a careful quantitative and experimental analysis.

We believe that in a similar way it will be possible to go beyond, re-visiting pseudo-boolean constraints and other important constraints that are well known in the Constraint Programming community.

References

[Bat68] Batcher, K.E.: Sorting Networks and their Applications. In: AFIPS Spring Joint Computing Conference, pp. 307–314 (1968)

[BB03] Bailleux, O., Boufkhad, Y.: Efficient CNF encoding of Boolean Cardinality Constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 108–122. Springer, Heidelberg (2003)

[DLL62] Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-Proving. Communications of the ACM, CACM 5(7), 394–397 (1962)

[DP60] Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM, JACM 7(3), 201–215 (1960)

[ES06] Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2, 1–26 (2006)

[MSP08] Marques-Silva, J., Planes, J.: Algorithms for Maximum Satisfiability usint Unsatisfiable Cores. In: DATE 2008, pp. 408–413. IEEE Computer Society Press, Los Alamitos (2008)

[Sin05] Sinz, C.: Towards an optimal CNF encoding of boolean cardinality constraints. In: van Beek, P. (ed.) CP 2005, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)

[SL07] Marques-Silva, J.P., Lynce, I.: Towards robust CNF encodings of cardinality constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 483–497. Springer, Heidelberg (2007)

New Encodings of Pseudo-Boolean Constraints into CNF

Olivier Bailleux[1], Yacine Boufkhad[2], and Olivier Roussel[3]

[1] LERSIA – Université de Bourgogne
olivier.bailleux@u-bourgogne.fr
[2] LIAFA, CNRS, Gang team INRIA, Université Paris Diderot, France
boufkhad@liafa.jussieu.fr
[3] Université Lille-Nord de France, Artois, F-62307 Lens – CRIL, F-62307 Lens – CNRS UMR 8188, F-62307 Lens
olivier.roussel@cril.univ-artois.fr

Abstract. This paper answers affirmatively the open question of the existence of a polynomial size CNF encoding of pseudo-Boolean (PB) constraints such that generalized arc consistency (GAC) is maintained through unit propagation (UP). All previous encodings of PB constraints either did not allow UP to maintain GAC, or were of exponential size in the worst case. This paper presents an encoding that realizes both of the desired properties. From a theoretical point of view, this narrows the gap between the expressive power of clauses and the one of pseudo-Boolean constraints.

Keywords: Pseudo-Boolean, SAT translation.

1 Introduction

Many practical problems can be expressed as constraint satisfaction problems and several formalisms for constraint satisfaction have been defined: integer linear programming (ILP) [7], constraint satisfaction problems (CSP), pseudo-Boolean constraints (PB) [2], propositional satisfiability (SAT) and Quantified Boolean Formulae (QBF) to name only a few. These formalisms differ by the expressivity of their constraints, the power of the inference rules that can be used and the efficiency of the corresponding solvers.

Complexity theory is a first approach to compare these formalisms. For instance, QBF has a higher complexity than SAT, PB, CSP and ILP. Both SAT, PB, CSP[1] and ILP are NP-complete problems. This implies that, from the point of view of complexity theory, all these formalisms have the same expressive power in the sense that there exist polynomial reductions from any of these problems to another.

The existence of a polynomial reduction from a formalism F to another formalism F' means that a problem of F can always be solved by translating it into

[1] Under the assumption that each constraint of the CSP can be checked in polynomial time.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 181–194, 2009.

a problem of F', and that such an indirect approach only implies a polynomial overhead which can be considered as negligible for NP problems. This approach can also be quite interesting in practice if solvers for F' are more efficient than solvers for F. The efficiency of modern SAT solvers explains why SAT has become very popular to encode and solve a number of problems. Some examples are Model checking [6], Symbolic reachability [1], Planning [8], Scheduling [12], Diagnosis [10], etc.

There may exist many encodings of a given problem in SAT and these encodings are not equivalent regarding SAT solving methods. For example, in [11] it is shown that a problem can be encoded to SAT in two ways, one that is exponential for resolution and another which is polynomial although the two encodings are of polynomial size. Similarly in [4], some difficult benchmarks for SAT solvers are shown to be easy if encoded in a different way. Thus, to make the polynomial reduction to SAT a practical approach for solving problems, the question is to find a polynomial reduction that preserves the basic inferences in a given problem through the basic inferences used in SAT solvers. More precisely, this paper focuses on Unit Propagation as the basic mechanism of inference in complete SAT solvers.

For example, it is known that maintaining Arc Consistency on CSP instances defined with only constraints in extension is equivalent to applying Unit Propagation on a polynomial SAT encoding of the constraints named "support encoding" [9]. Another encoding with the same property is proposed in [3]. These encodings show that SAT and CSP with only extensional constraints are very close problems. The same approach is used in this paper to compare the pseudo-Boolean formalism with the Satisfiability formalism.

Pseudo-Boolean constraints can be seen as an extension of the clauses of SAT, or as a special case of integer programming constraints. The following observations suggest that pseudo-Boolean constraints are stronger than SAT constraints (clauses). While clauses can be considered as constraints defined in extension (a clause forbids one single partial assignment), pseudo-Boolean constraints are defined in intention through a mathematical formula. Another observation is that a SAT encoding of a PB constraint which doesn't introduce extra variables requires an exponential number of clauses in the worst case. Besides, so far, all known SAT encodings of a PB constraint were either of exponential size or did not allow unit propagation to maintain GAC. This article presents a polynomial SAT encoding of PB constraints which proves that the basic inferences on PB constraints (Generalized Arc Consistency) polynomially reduce to Unit Propagation in SAT. To the best of our knowledge, the existence of such an encoding was an open question. This result narrows the gap between the PB and SAT formalisms.

Section 2 introduces the definitions and notations used throughout this paper. The general idea of the new encoding is to sum the terms on the left side of the PB constraint and compare it to the right term. The property required for the encoding of the addition is that it must give a result even when some input variables are unassigned. To reach this goal, the encoding is based on a unary

representation of numbers (see section 3). For conciseness, coefficients in the constraints are decomposed in binary, and a unary representation is used for each power of two. A kind of carry between unary representations ensures that the most significant unary representation computes the sum of the coefficients of the true literals in the constraint. The key point in the encoding is to add a constant to each side of the PB constraint so that the right term becomes a multiple of a power of two, which makes the comparator trivial.

This general encoding is detailed in section 4.1. It can be declined in two different versions: GPW that detects inconsistencies (section 4.2) and LPW that maintains Generalized Arc Consistency (section 4.3). LPW is the first SAT encoding of a pseudo-Boolean constraint of n variables with a maximum coefficient of a_{max} which is both polynomial ($O(n^2 \log(n) \log(a_{max}))$ variables and $O(n^3 \log(n) \log(a_{max}))$ clauses) and which lets Unit Propagation maintain Generalized Arc Consistency. Section 5 compares the new method with the previous encodings of PB constraints into SAT. At last, some perspectives are given.

2 Definitions and Notations

In this paper, we only consider constraints which are defined over a finite set of Boolean variables x_j. Boolean variables can take only two values 0 (*false*) and 1 (*true*). A literal l_j is either a Boolean variable x_j or its negation $\overline{x_j}$ (with $\overline{x_j} = 1 - x_j$). A *linear pseudo-Boolean constraint* is a constraint over Boolean variables defined by $\sum_j a_j l_j \triangleright M$ where a_j and M are integer constants, l_j are literals and $\triangleright$ is one of the classical relational operators ($=, >, \geq, <$ or $\leq$). Without loss of generality, these constraints can be rewritten to use only the less operator and positive coefficients a_j (since $-a.x$ can be rewritten as $a.\overline{x} - a$). A *clause* is a disjunction of literals. A clause $l_1 \vee l_2 \vee \ldots \vee l_n$ is equivalent to $l_1 + l_2 + \ldots + l_n \geq 1$. So clauses are a special case of pseudo-Boolean constraints where each $a_j = 1$ and $M = 1$. An assignment is a mapping of Boolean variables to their value (0 or 1). In the CSP context, an instantiation is a mapping of variables to a value in their domain.

Unit propagation (UP) is the fundamental mechanism used in most SAT solvers. Whenever each literal of a clause but one is false, the remaining literal must be set to true in order to satisfy the clause. The derivation of a literal l by unit propagation from formula f will be denoted $f \vdash_{UP} l$. The derivation of the empty clause is denoted $f \vdash_{UP} \bot$. *Generalized Arc Consistency* (GAC) is one of the fundamental inference rules in CSP. Let $scp(C)$ denote the scope of a constraint C, which is the set of variables constrained by C. A value a of a variable X is generalized arc consistent in a constraint C if $X \notin scp(C)$ or, when $X \in scp(C)$, if there exists an instantiation I of the other variables in the scope of C such that $I \cup \{X = a\}$ satisfies C. A value a of X is generalized arc consistent if it is generalized arc consistent in every constraint. A CSP instance is generalized arc consistent if each value of each variable is generalized arc consistent. Enforcing GAC consists in removing values which are not generalized arc consistent from the domain of their variable. For example, let us consider the

PB constraint $x_1 + 2x_2 + 4x_3 < 3$, value 1 for x_3 is not generalized arc consistent because no assignment of x_1, x_2 can satisfy the constraint once $x_3 = 1$. Enforcing GAC on this constraint will remove 1 from the domain of x_3 and hence assign 0 to x_3 .

An *encoding* E of a source language L_S to a target language L_T is a mapping of constraints of L_S to sets of constraints of L_T such that any formula f of L_S is equivalent to $E(f)$ in a generalized sense: any model of f can be extended to obtain a model of $E(f)$ (and conversely any model of $E(f)$ can be projected on the vocabulary of f to get a model of f). In the following definitions, we only consider languages where variables are Boolean[2]. An encoding E is said to *UP-detect inconsistency* if, for any constraint C of the source language and any assignment A of the source language, $C \wedge A \models \bot \Leftrightarrow E(C) \wedge A \vdash_{UP} \bot$. An encoding E is said to *UP-maintain GAC* if, for any constraint C, any assignment A and any literal l of the source language, $C \wedge A \models l \Leftrightarrow E(C) \wedge A \vdash_{UP} l$.

For example, let us consider the pseudo-Boolean constraint $x_1 + 2x_2 + 4x_3 < 6$. Given the partial assignment $\{x_1 = 1, x_3 = 1\}$, any encoding which UP-maintains GAC will allow unit propagation to fix $x_2 = 0$. Given the partial assignment $\{x_2 = 1, x_3 = 1\}$, any encoding which UP-detects inconsistency must allow unit propagation to produce the empty clause.

Obviously, all things being equal, the more a solver propagates, the more efficient it is. On the other hand, encodings which UP-maintain GAC generally produce larger formulae than the other ones because they must encode each potential implication of a literal. Of course, larger formulae slow down unit-propagation. It is then not always clear which is the best trade-off between the size of encodings and their ability to enforce propagations.

3 Unary Representation and Cardinality Constraints

First, let us recall briefly the notion of unary representation of integer intervals. The details are in [4].

An integer variable u taking its values in the range $0..k$ is represented by a vector of k Boolean variables $U = \langle u_1, ..., u_k \rangle$. At any time, only variables on the left of this vector can be assigned 1, only variables on the right can be assigned 0 and variables in between are unassigned. More formally, U takes its values in a set $\mathcal{U}_k \subset \{0, 1, *\}^k$ ($*$ standing for unassigned) such that there exists two ranks a and b ($0 \leq a \leq b \leq k$) having the following property: $u_i = 1$ if $i \leq a$, $u_i = *$ if $a < i \leq b$ and $u_i = 0$ if $i > b$.

An integer u such that $u = m$ is represented by the vector having $u_1 = u_2 = ... = u_m = 1$ and $u_{m+1} = ... = u_k = 0$. The advantage of unary vectors is that they allow the representation of integer intervals. For example $a \leq u \leq b$ is represented by a vector that assigns 1 to the a first Boolean variables and 0 to the $k - b$ last ones, the remaining variables being unassigned.

[2] However, the generalization to languages where a variable X can take multiple values v_i is straightforward. For example, an encoding E UP-maintains GAC if, $\forall C, \forall A, \forall X, \forall v$, $C \wedge A \models X \neq v \Leftrightarrow E(C) \wedge E(A) \vdash_{UP} E(X \neq v)$.

The other advantage of this representation is that it allows to encode an addition in such a way that unit propagation is able to do the expected inferences, even when some variables are unassigned. Let U and V be two unary vectors representing respectively two integers u and v and let W be the unary representation of their sum $w = u + v$. The encoding of this addition contains clauses of the type $\overline{u_a} \vee \overline{v_b} \vee w_{a+b}$ stating that whenever $u \geq a$ and $v \geq b$ for some values a and b then $w \geq a + b$. We will denote by $\psi(U \oplus V = W)$ the conjunction of all the clauses of that type that ensure that $w \geq u + v$ through their unary representations. The sum of integers is naturally extended to the sum of their representations through the operator $\oplus$. Formally, for two unary vectors $U = \langle u_1, u_2, .., u_k \rangle$, $V = \langle v_1, v_2, .., v_l \rangle$ and $W = U \oplus V = \langle w_1, w_2, .., w_{k+l} \rangle$ with the convention $u_0 = v_0 = w_0 = 1$:

$$\psi\left(W = U \oplus V\right) = \bigwedge_{a=0}^{k} \bigwedge_{b=0}^{l} (\overline{u_a} \vee \overline{v_b} \vee w_{a+b})$$

Some other clauses ensuring that $w \leq u + w$ are needed to obtain the encoding of [4] but are omitted because they are not relevant in this paper. Clearly the number of clauses in $\psi\left(W = U \oplus V\right)$ is $O(n^2)$ when the numbers are of size n.

In [4], the unary representation is used to efficiently encode cardinality constraints. The vector of variables involved in the cardinality constraint called input variables is connected to a unary vector called output vector representing its number of 1s through a CNF formula called a Totalizer. The Totalizer is in charge of transforming the input vector in an output vector which contains the same values but which also satisfies the requirements of the unary representation (all 1s on the left, all 0s on the right and all unassigned variables in the middle). In essence, this Totalizer plays the same role as a sorting network.

The Totalizer used in the encoding schemes described in this paper is simpler than the one used in [4] because we never use the 0s in the output. All is needed is that the Totalizer generate an output vector with all 1s on the left (as many as in the input vector) and all other variables unassigned. For any vector X of Boolean variables, let $U(X)$ denote the vector of the unary representation of the number of its 1s as enforced by the Totalizer. Let $\Phi(X)$ be the Totalizer which transforms X into $U(X)$. It is built in a recursive manner as described in the recursive procedure $\Phi(X)$ of Algorithm 1. Indeed, the unary representation of a single variable is the variable itself, the unary representation of the number of 1s of vector $X = \langle x_1, x_2, ..., x_n \rangle$ is the sum of the unary representations of $X_1 = \langle x_1, x_2, ..., x_{\lfloor n/2 \rfloor} \rangle$ and $X_2 = \langle x_{\lfloor n/2 \rfloor + 1}, ...x_n \rangle$, and the CNF formula enforcing this conjunction is $\psi(U(X_1) \oplus U(X_2) = U(X))$. The whole formula of the totalizer of some vector X is denoted by $\Phi(X)$. It is the conjunction of the formulae ψ.

The fundamental property of the formula $\Phi(X)$ as it will be used later is that for any partial assignment to the variables X unit propagation enforces $U(X)$ to be the unary representation of the number of ones in X. The number of variables created by the encoding is clearly $O(n \log(n))$ and the number of clauses is $O(n^2 \log(n))$ since the procedure makes $O(log(n))$ recursive calls.

Algorithm 1. $\Phi(X)$

Require: A vector $X = \langle x_1, x_2, ..., x_k \rangle$
  if $k = 1$ **then**
    $U(X) \leftarrow \langle x_1 \rangle$
    return true
  else
    $U(X) \leftarrow \langle u_1, u_2, ..., u_k \rangle$ {u_i are obtained from a global unique variable generator}

    $X_1 \leftarrow \langle x_1, x_2, ..., x_{\lfloor n/2 \rfloor} \rangle$
    $X_2 \leftarrow \langle x_{\lfloor n/2 \rfloor + 1}, ...x_n \rangle$
    return $\Phi(X_1) \wedge \Phi(X_2) \wedge \psi(U(X_1) \oplus U(X_2) = U(X))$
  end if

In the rest of the description of the encoding, it is necessary to define an operator $\frac{1}{2}$ on the vectors of unary representations of integers such that for some vector $W = \langle w_0, w_1, ..., w_{2i}, w_{2i+1}, ... \rangle$, $W^{\frac{1}{2}} = \langle w_1, w_3, ..., w_{2i+1} ... \rangle$ is the vector of variables of odd ranks in the original. Clearly if W is the unary representation of some integer w then $W^{\frac{1}{2}}$ is the unary representation of $\lfloor \frac{w}{2} \rfloor$.

4 Global and Local Polynomial Watchdog Encoding Schemes

We present in this section two SAT encoding schemes LPW and GPW standing respectively for *Local Polynomial Watchdog* and *Global Polynomial Watchdog*. A watchdog is a formula which will set a Boolean variable to 1 as soon as a constraint gets falsified. LPW UP-maintains GAC while producing formulae of polynomial size. GPW, which UP-detects inconsistencies, is more of practical interest since it produces smaller formulae at the cost of losing the property of UP-maintaining GAC.

4.1 Polynomial Watchdog

In the following we will consider without loss of generality a unique constraint C defined by the sequence of positive integer coefficients $(a_i)_{i=1..n}$ and an integer M constraining $\sum_{i \in I} a_i x_i < M$ where $I = \{1, 2, ..., n\}$ is a set of indices ranging from 1 to n the number of Boolean variables involved in this constraint. We consider only non trivial constraints i.e $\sum_{i \in I} a_i > M$. For some integer a, let $b_j(a)$ be the value of the bit of rank j in the binary representation of a.

A polynomial watchdog (PW) associated with a constraint C is a CNF formula denoted by $PW(C)$ based on the following sets of variables: the input variables $\{x_i | i \in I\}$ of the constraint C, and a set of additional variables called encoding variables. $PW(C)$ has a single output variable ω. The formula $PW(C)$ is built in such a way that it has the following property: for every partial assignment to the input variables that violates the constraint C, unit propagation applied to $PW(C)$ assigns the value 1 to the output variable ω.

The idea used to construct $PW(C)$ is to decompose each coefficient of the constraint in its binary representation and sum each bit having the same weight 2^k in a single Totalizer. There are as many Totalizers as the number of bits of the greatest coefficient. Half of the value of the totalizer for weight 2^k is computed with operator $\frac{1}{2}$ and integrated in the totalizer for weight 2^{k+1} (this is a kind of carry). The value represented by the different Totalizers must be compared to M. To make this comparison trivial, the constraint is first rewritten so that the right term becomes a multiple of the weight of the last Totalizer. Once this is done, the value of the comparison is represented by one single bit of the last Totalizer. All computations can be performed by unit propagation, even when some input variables are unassigned.

Let us now detail how the formula $PW(C)$ is built. The binary representation of the coefficients is considered. Let p be the index of the most significant bit in the greatest a_i. In other words, p is the integer such that $p+1$ is the number of bits necessary to represent the largest coefficient, namely $p = \lfloor \log_2(max_{i=1..n}(a_i)) \rfloor$.

An important feature used by the Polynomial Watchdog encoding is the tare which is an integer denoted by T. It turns out that the comparison with the right side of the constraint is trivial when it is a multiple of 2^p. For this reason, we define the tare T as the smallest integer such that $M + T$ is a multiple of 2^p. Let m denote the integer such that $M + T = m2^p$ and let $t_{p-1}...t_1t_0$ denote the binary representation of T over $p-1$ bits ($T < 2^p$). Considering this, the constraint can be rewritten to an equivalent form $T + \sum_{i \in I} a_i x_i < m2^p$.

For every j such that $0 \leq j \leq p$, let B_j be the set containing the input variables with the bit of rank j equal to 1 in the binary representation of their coefficient plus the constant t_j (the jth bit of the tare) if $t_j = 1$. More formally $B_j = \{x_i | b_j(a_i) = 1\} \cup \{t_j \text{ if } t_j = 1\}$ for $0 \leq j \leq p$. The sets B_j are called buckets.

Example 1. For the constraint $2x_1 + 3x_2 + 5x_3 + 7x_4 < 11$, we have $p = 2$, $T = 1$ ($t_0 = 1$, $t_1 = 0$) and the buckets are $B_0 = \{1, x_2, x_3, x_4\}$, $B_1 = \{x_1, x_2, x_4\}$ and $B_2 = \{x_3, x_4\}$. Figure 1 represents the different buckets and generated circuits.

The formula $PW(C)$ is built recursively by cascading $p+1$ Totalizers (see Section 3). Let $PW_j(C)$ denote the Totalizer number j and S_j denote its output. The CNF encoding of the totalizers and their inputs are defined recursively as follows. $\langle B_j \rangle$ is a vector formed by the elements of the bucket B_j taken in an arbitrary order:

- For $j = 0$, let $PW_0(C) = \Phi(\langle B_0 \rangle)$. The output is $S_0 = U(\langle B_0 \rangle)$.
- For any $1 \leq j \leq p$, $PW_j(C) = \Phi(\langle B_j \rangle) \wedge \psi(S_j = U(\langle B_j \rangle) \oplus S_{j-1}^{\frac{1}{2}})$. The output unary vector is $S_j = U(\langle B_j \rangle) \oplus S_{j-1}^{\frac{1}{2}}$ enforced through the formula $\psi(S_j = U(\langle B_j \rangle) \oplus S_{j-1}^{\frac{1}{2}})$.

The polynomial watchdog of the constraint C can now be defined as: $PW(C) = \bigwedge_{j=0}^{p} PW_j(C)$ The mth variable of the vector S_p is the output variable ω (S_p has at least m bits because the constraint is not trivial). The algorithm 2 describes the steps in the computation of the formula $PW(C)$ and Figure 1 shows $PW(C)$ on the constraint of Example 1.

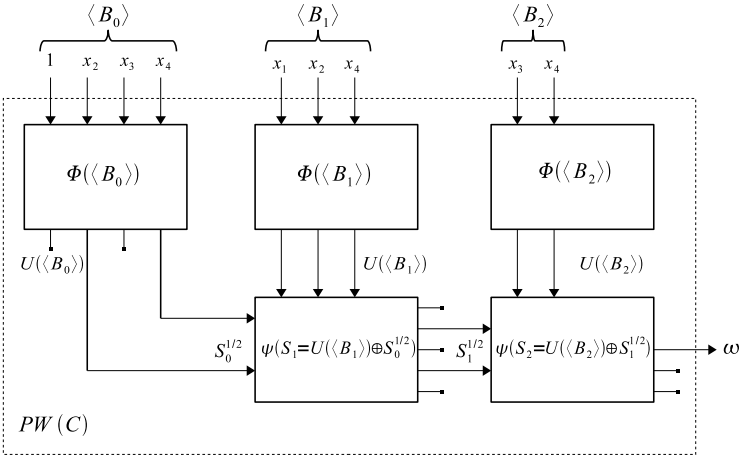


Fig. 1. Schematic representation of PW($2x_1 + 3x_2 + 5x_3 + 7x_4 < 11$)

Algorithm 2. PW(C)

Require: a constraint $\sum_{i=1}^{n} a_i x_i < M$
  $p \leftarrow \log_2(max_{i=1..n}(a_i))$
  $T \leftarrow (m2^p - M)$ s.t. m is the smallest integer having $m2^p \geq M$
  for $j = 0$ to p **do**
    $B_j \leftarrow \{x_i | b_j(a_i) = 1\} \cup \{t_j \text{ if } t_j = 1\}$
  end for
  $F \leftarrow \Phi(\langle B_0 \rangle)$
  $S_0 \leftarrow U(\langle B_0 \rangle)$
  for $j = 1$ to p **do**
    $F \leftarrow F \wedge \Phi(\langle B_j \rangle)$
    $S_j \leftarrow U(\langle B_j \rangle) \oplus S_{j-1}^{\frac{1}{2}}$
    $F \leftarrow F \wedge \psi(S_j = U(\langle B_j \rangle) \oplus S_{j-1}^{\frac{1}{2}})$
  end for
  return F

A polynomial watchdog is based on $p+1$ Totalizers requiring each $O(n \log(n))$ variables and $O(n^2 \log(n))$ clauses. Then, in general, a constraint involving n Boolean variables and having coefficients of at most a_{max} generates at most $O(n \log(n) \log(a_{max}))$ variables and $O(n^2 \log(n) \log(a_{max}))$ clauses.

Lemma 1. *For any partial assignment to the variables of C, Unit Propagation on $PW(C)$ assigns 1 to ω if and only if this partial assignment is inconsistent with C.*

Proof. Let s_i be the number of 1s in bucket B_i whose unary representation is $U(\langle B_i \rangle)$. The lemma follows from the fact that UP enforces at any time S_i for $0 \leq i \leq p$ to be the unary representation of $\lfloor \frac{\sum_{j=0}^{i} s_j 2^j}{2^i} \rfloor$. In particular, for p,

since $\lfloor \frac{\sum_{j=0}^{p} s_j 2^j}{2^p} \rfloor = \lfloor \frac{T+\sum_{j=0}^{p} a_j x_j}{2^p} \rfloor$, UP assigns 1 to the mth bit ω if and only if the left side is greater or equal to $M+T$ and hence the constraint is violated.

The fact that S_i for $0 \leq i \leq p$ is the unary representation of $\lfloor \frac{\sum_{j=0}^{i} s_j 2^j}{2^i} \rfloor$ can be proven by induction on i. For $i = 0$, it is obviously true thanks to the first Totalizer. Suppose now that the property is true for some i i.e. it is true that UP enforces S_i to be the unary representation of $\lfloor \frac{\sum_{j=0}^{i} s_j 2^j}{2^i} \rfloor$, since $S_{i+1} = U(\langle B_{i+1} \rangle) \oplus S_i^{\frac{1}{2}}$, UP will enforce — through the Totalizer $\Phi(\langle B_{i+1} \rangle)$ and $\psi(S_{i+1} = U(\langle B_{i+1} \rangle) \oplus S_i^{\frac{1}{2}})$ — S_{i+1} to be the unary representation of $s_{i+1} + \lfloor \frac{\lfloor \frac{\sum_{j=0}^{i} s_j 2^j}{2^i} \rfloor}{2} \rfloor = s_{i+1} + \lfloor \frac{\sum_{j=0}^{i} s_j 2^j}{2^{i+1}} \rfloor = \lfloor \frac{\sum_{j=0}^{i+1} s_j 2^j}{2^{i+1}} \rfloor$.

4.2 Global Polynomial Watchdog

The Global Polynomial Watchdog (GPW) is an encoding that detects inconsistencies. It is based on the PW described in the previous section. It consists simply in adding the unit clause $\overline{w}$. The formula generated to encode a constraint C is then $GPW(C) = PW(C) \wedge \overline{w}$. GPW has the same complexity than $PW(C)$.

Proposition 1. *A partial assignment of the variables of a constraint C is inconsistent with it if and only if unit propagation applied to $GPW(C)$ detects an inconsistency.*

However GPW does not UP-maintain GAC. This can be seen in the following counterexample. GAC on the partial assignment $x_3 = 1$ on Example 1 assigns $x_4 = 0$ but UP will not detect such an assignment. Indeed, all what UP will derive is that the higher bits of $U(\langle B_2 \rangle)$ and $S_1^{\frac{1}{2}}$ cannot be equal to 1 at the same time. This situation is obtained if $x_4 = 1$ but UP cannot foresee it.

Although GPW does not UP-maintain GAC it is not far from doing it. Indeed, GAC can be maintained through UP look-ahead. A UP look-ahead consists in trying to assign 1 to an unassigned input variable and then to run UP. If an inconsistency is detected, the variable must be assigned 0 otherwise it remains unassigned. GAC can be maintained through UP look-ahead on the GPW encoding but few modern SAT solvers perform such tests.

4.3 Local Polynomial Watchdog

This encoding is analogous to the support encoding [9] applied to pseudo-Boolean constraints. Each variable x of the encoded constraint is connected to a watchdog formula that assigns through unit propagation $x = 0$ when the value $x = 1$ has no support.

Consider the constraint $\sum_{i \in I} a_i x_i < M$. Let $I_k = I \backslash \{k\}$ and let C_k be the constraint defined as $\sum_{i \in I_k} a_i x_i < M - a_k$. Clearly, whenever the constraint C_k is inconsistent with a partial assignment, the variable x_k must be fixed to 0.

Consider $PW(C_k)$ the polynomial watchdog encoding of the constraint C_k and ω_k the output variable of this encoding. As described by Algorithm 3,

Algorithm 3. $LPW(C)$

Require: a constraint $\sum_{i=1}^{n} a_i x_i < M$
  $F \leftarrow$ **true**
  for $k = 1$ to n **do**
    $C_k \leftarrow \sum_{i=1..n, i \neq k} a_i x_i < M - a_k$
    $F \leftarrow F \wedge (PW(C_k) \wedge (\overline{\omega_k} \vee \overline{x_k}))$
  end for
  return F

the Local Polynomial Watchdog can now be defined as the CNF: $LPW(C) = \bigwedge_{k=1}^{n} (PW(C_k) \wedge (\overline{\omega_k} \vee \overline{x_k}))$.

Theorem 1. *Any pseudo-Boolean Constraint of integer weights using n variables having a maximum weight of a_{max} can be translated into a CNF formula of $O(n^2 \log(n) \log(a_{max}))$ variables and $O(n^3 \log(n) \log(a_{max}))$ clauses such that Unit Propagation maintains Generalized Arc Consistency.*

Proof. The proof follows from the fact that GAC assigns 0 to some x_k if and only if the corresponding C_k is inconsistent with the partial assignment. In this case UP assigns $\omega_k = 1$ and then $x_k = 0$. The complexity comes from the fact that we have n Polynomial Watchdogs.

4.4 Implementation

The size of the watchdogs used in the two proposed encodings can be reduced by sharing sub-formulae – both into the same watchdog as well as between different ones – in a way to reduce the number of clauses. It is even possible to share sub-formulae between several input constraints. The key of such an optimization is how to split the input variables of each totalizer into the two sets of input variables of its sub-totalizers. A first basic implementation has been done, for validation purpose only. It is based on a static ordering of the variables of the input constraint, which are sorted in decreasing order of their coefficients. No extensive experimental evaluations of the LPW end GPW encodings has been performed yet because the implementation is not yet optimized, and anyway an extensive evaluation is not the purpose of this paper[3].

That said, the first few results suggest that the new encodings could be of practical interest. For exemple, the following unsatisfiable Bin-packing instance was encoded using our basic implementation: 16 objects with weights 211, 203, 202, 201, 200, 199, 198, 197, 196, 194, 175, 167, 166, 165, 164, and 162 must be put into 3 boxes, each with capacity 1000. For each box i and each object j, a Boolean variable x_{ij} denotes whether the object i belongs to the box j.

[3] Some tests were done on 1D bin packing instances, randomly generated instances and hand crafted instances, with the only aim to verify that the size of the LPW and GPW encodings does not make them intractable. We do not have enough space to present these experiments.

Three pseudo-Boolean constraints ensure the capacity requirement of each box, and 16 additional cardinality constraints ensure that each object belongs exactly to one box. The BDD encoding of [5] (see section 5) produces 38077 literals in 15637 clauses, and allows `minisat` to solve the problem within 486 seconds; the LPW encoding produces 58521 literals in 21615 clauses, and allows `minisat` to solve it within 12 seconds; the GPW encoding produces 7108 literals in 2714 clauses, and allows `minisat` to solve it within 3 seconds; the pseudo-Boolean solver `pueblo` [14] solves the initial instance within 23 seconds; `minisat+` solves the initial instance within 8 seconds.

In some cases, our *basic* version of the LPW encoding seems to produce a prohibitive number of clauses. For exemple, to put 50 objects into 5 boxes, each with capacity 1000, it required 2553715 literals in 884945 clauses, while GPW produced "only" 95675 literals in 34200 clauses.

5 Related Work

In [16], Warner proposes a linear CNF encoding of pseudo-Boolean constraints. It uses a binary adder network, which does not allow unit propagation to detect whether any input constraint is falsified by a given partial assignment.

[4] proposes an encoding which UP-maintains GAC on cardinality constraints. It is based on an extended version of the totalizers described in section 3 and requires $O(n \log n)$ additional variables and $O(n^2)$ clauses of size at most 3 to encode a constraint with n variables. [13] also introduces an encoding based on a unary representation but uses a odd-even merge sorting network, thereby reducing the number of clauses to $O(n \log^2 n)$.

In [15], Sinz introduces two other encodings. The first one uses a sequential adder network with a unary representation of integers. It maintains arc-consistency and, given a cardinality constraint $\sum_{j=1}^{n} x_i < M$, it produces a formula of size $O(nM)$. The second one uses a parallel adder network with a binary representation of integers. It does not detects local inconsistencies and produces a formula of size $O(n)$, but smaller than the one produced by Warner's encoding.

In 2006, Eén and Sörensson released the pseudo-Boolean solver `minisat+` [13], one of the best performers in the PB'06 competition. It is based on a conversion of pseudo-Boolean constraints to a CNF formula, which is submitted to the `minisat` solver. `minisat+` uses some heuristics to choose between three encoding techniques based respectively on binary decision diagram (BDD), adder networks, and sorting network. Another variant of BDD based encoding is simultaneously (and independently) introduced in [5].

The BDD based encoding transforms each pseudo-Boolean constraint into a binary decision diagram. Each node in the BDD represents a pseudo-Boolean constraint and the satisfaction of this constraint is reified by a propositional variable in the encoding. The root of the BDD represents the constraint to encode. Each node has at most two children which are obtained by assigning the first variable of the constraint to the two possible values 0/1 and simplifying

the resulting constraints. Nodes corresponding to trivial constraints are pruned. Two nodes can also share a common child. The relation between the truth of a node and the truth of its children only depends on the variable chosen to decompose the constraint and the variables corresponding to the nodes. In [13], this relation is encoded in six ternary clauses. In [5], a slightly different encoding is used, which translates each node of the BDD into two binary clauses and two ternary clauses. These two encodings maintain arc-consistency, but can produce an exponential number of clauses in the worst case [5].

The encoding based on adder networks produces a number of clauses (of length at most 4) linearly related to the size of the encoded constraint, as [16], but using a different structure. All the variables with a bit of a given weight in the base 2 representation of their coefficients are bundled in a *bucket*. The number of bits set to 1 in each bucket is computed using a binary adders network. The results are then combined thanks to additional adders. The resulting binary value feeds a comparator, which is optimized to deal with the constant bound of the constraint. Like Warner's one, this linear encoding does not detect inconsistencies, then cannot maintain arc-consistency.

In [13], the encoding based on sorting networks is founded on the unary representation of numbers [4,13] (see section 3). To compress the representation, a number is represented by several buckets in unary notation and each bucket has its own weight. Instead of using weights which are a power of a base b $(1, b, b^2, b^3, ..)$, [13] uses a general increasing sequence of positive integers. This is in fact a generalization of the usual representation of numbers in a base b with the exception that the ratio of the weights of two successive digits is no more a constant. The GPW encoding presents similarities with this encodings but there are several differences: (1) each of our bucket is related to a power of two, while the encoding used in `minisat+` uses arbitrary weights; (2) instead of our totalizers, `minisat+` uses odd even merge sorting networks; (3) `minisat+` does not uses a tare, which is an essential feature of GPW to ensure that the bound of the constraint is a round number of the weight of the last sorter. As a consequence, instead of a simple unit clause, a more complex non monotone circuit is used to establish whether the constraint is satisfied or not. More importantly, it does not maintain arc-consistency and it is not proved in [13] whether it UP-detects inconsistency or not.

Let us mention the standard exponential transformation with no auxiliary variables. Except for the trivial cases (i.e., constraints with only one literal and constraints that are either impossible to satisfy or to falsify), a constraint $\sum_{i=1}^{n} a_i x_i < M$ is translated as two sets of clauses. The first one encodes $(x_n = 0) \vee (\sum_{i=1}^{n-1} a_i x_i < M - a_n)$, and the second one encodes $\sum_{i=1}^{n-1} a_i x_i < M$. Unit resolution achieves the same propagations in this encoding as it does in the BDD one, then it maintains arc-consistency and detects local inconsistencies. Contrarily to the BDD one, direct encoding does not require additional variables, but it often produces bigger formulae because each clause corresponds to a *path* of the BDD. However, small constraints (with typically less than 6 variables) tend to produce more concise formulae.

6 Synthesis and Perspectives

This paper provides a theoretical contribution on the encoding of pseudo-Boolean constraints into CNF formulae. Now, it is known that there exists a polynomial encoding that allows unit propagation (implemented in all DPLL-based SAT solvers) to restore generalized arc consistency on the initial constraints. Clearly, this result opens new questions and new perspectives in the field of indirect resolution of pseudo-Boolean problems.

The space complexity of the proposed encoding is $O(n^3 \log(n) \log(a_{max}))$ but in practice many sub-formulae could be shared by several totalizers – both into the same watchdog as well as among different ones – in a way to reduce the number of produced clauses. The key is the order of the variables into the vector related to each bucket $\langle B_j \rangle$. Some work must be done to search for relevant ordering heuristics.

Another issue is the existence of structurally more concise encodings that maintain arc consistency. The underlying theoretical aim is to establish the minimum size for any such encoding. At this point, it is not known if there is a gap – in terms of space complexity[4] – between encodings which "only" detect local inconsistencies and encodings maintaining arc consistency. Furthermore, it is not clear whether the former outperforms the latter with modern SAT solvers. Answering these questions will probably require an extensive amount of future work.

Moreover, the existing encodings could be combined in a way to use a specific encoding for each individual constraint (and even the possibility to use redundant encodings could be considered). The `minisat+` solver always uses such an approach, but it may be improved and extended to the new encodings introduced in the present paper. These new encodings could also be improved by using a sorting network, as in [13], instead of a totalizer.

An even more crucial question is whether solving pseudo-Boolean problems with SAT solvers is actually relevant. On the one hand, this approach proved its efficiency (see `minisat+` at PB05) despite the fact that state-of-the-art encodings are not mature. On the other hand, one can hardly expect a SAT solver to outperform a native pseudo-Boolean solver when this technology becomes mature.

References

1. Abdulla, P.A., Bjesse, P., Eén, N.: Symbolic reachability analysis based on SAT-solvers. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 411–425. Springer, Heidelberg (2000)
2. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Generic ILP versus Specialized 0-1 ILP: An Update. In: Proc. of Int. Conf. on Computer Aided Design (ICCAD 2002), pp. 450–457 (2002)
3. Bacchus, F.: Gac via unit propagation. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 133–147. Springer, Heidelberg (2003)

[4] Regarding the degree of the polynomial.

4. Bailleux, O., Boufkhad, Y.: Efficient CNF encoding of boolean cardinality constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 108–122. Springer, Heidelberg (2003)
5. Bailleux, O., Boufkhad, Y., Roussel, O.: A Translation of Pseudo Boolean Constraints to SAT. Journal on Satisfiability, Boolean Modeling and Computation 2, 191–200 (2006)
6. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic Model Checking using SAT procedures instead of BDDs. In: Proc. of Design Automation Conference (DAC 1999), pp. 317–320 (1999)
7. Brand, S., Duck, G.J., Puchinger, J., Stuckey, P.J.: Flexible, Rule-based Constraint Model Linearisation. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 68–83. Springer, Heidelberg (2008)
8. Ernst, M., Millstein, T., Weld, D.S.: Automatic SAT-Compilation of Planning Problems. In: IJCAI 1997, pp. 1169–1176 (1997)
9. Gent, I.P.: Arc Consistency in SAT. In: Proc. of the Fifteenth European Conference on Artificial Intelligence (ECAI 2002), pp. 121–125 (2002)
10. Grastien, A., Anbulagan, A., Rintanen, J., Kelareva, E.: Diagnosis of Discrete-Event Systems Using Satisfiability Algorithms. In: AAAI-2007, pp. 305–310 (2007)
11. Hertel, A., Hertel, P., Urquhart, A.: Formalizing Dangerous SAT Encodings. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 159–172. Springer, Heidelberg (2007)
12. Baker, A.B., Crawford, J.M.: Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In: Proc. of the Twelfth National Conference on Artificial Intelligence, pp. 1092–1097 (1994)
13. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2, 1–26 (2006)
14. Sheini, H.M., Sakallah, K.A.: Pueblo: a modern pseudo-Boolean SAT solver. In: Design, Automation and Test in Europe, 2005. Proc., pp. 684–685 (2005)
15. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)
16. Warners, J.P.: A linear-time transformation of linear inequalities into conjunctive normal form. Information Processing Letters 68(2), 63–69 (1998)

Efficient Term-ITE Conversion for Satisfiability Modulo Theories*

Hyondeuk Kim[1], Fabio Somenzi[1], and HoonSang Jin[2]

[1] University of Colorado at Boulder
[2] Cadence Design Systems
{Hyondeuk.Kim,Fabio}@Colorado.EDU,
hsjin@cadence.com

Abstract. This paper describes how *term-if-then-else* (*term-ITE*) is handled in Satisfiability Modulo Theories (SMT) and to decide *Linear Arithmetic Logic* (*LA*) in particular. Term-ITEs allow one to conveniently express verification conditions; hence, they are very common in practice. However, the theory provers of SMT solvers are usually designed to work on conjunctions of literals; therefore, the input formulae are rewritten so as to eliminate term-ITEs. The challenge in rewriting is to avoid introducing too many new variables, while avoiding as often as possible the exponential explosion that is frequent when a naive approach is applied. We propose a solution that is based on cofactoring and theory propagation, which often produces orders-of-magnitude speedups in several SMT solvers for LA problems.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers find increasing applications in areas like formal verification in which one needs to reason about complex Boolean combinations of numerical constraints. The most common approach to this problem leverages the efficiency of modern propositional satisfiability solvers that work on a propositional abstraction of the given formula. At the same time, they interact with theory solvers, which check conjunctions of literals for consistency and learn consequences (new lemmas) from them. This approach has come to be known as DPLL(T) [12].

Among the logics for which theory solvers have been developed in recent times, linear arithmetic is one of the most useful and well-researched. Many current solvers adopt some variant of the simplex algorithm. In particular, the backtrackable version of [3] fits well in the DPLL(T) scheme and has shown good results in practice for both integer and real-valued variables.

The Boolean dimension of many SMT instances, however, continues to pose a challenge to solvers. In this paper we address this problem. In particular, we focus on those instances that make extensive use of the *term-if-then-else* (ITE) operator. This operator facilitates the analysis of problems in which paths through control-flow graphs must be translated into SMT formulae. It is not surprising, therefore, that many of the available benchmark instances for linear arithmetic are rich in term-ITEs. Given a code fragment

* This work was supported in part by SRC contract 1859-TJ-2008.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 195–208, 2009.

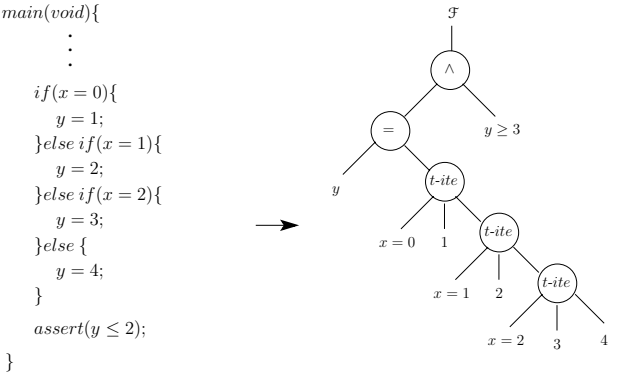


Fig. 1. Verification condition $\mathcal{F}$ with term-ITEs

that contains *if* statements, a verification condition can be naturally formulated with ITEs as shown in Fig. 1.

Two major approaches can be envisioned to deal with term-ITEs. On the one hand, one can modify the theory solver to deal with conditional expressions. Without ITEs, every assignment to an atom of the SMT formula adds to a conjunction of literals that is analyzed by the theory solver. With ITEs, this is no longer the case. In order to analyze the atom, the conditional expressions of the ITEs need to be assigned. On the other hand, one can eliminate all the ITEs from the formula by rewriting. The problem here is that the rewritten formula may retain a lot of redundancies depending on how one rewrites it. We address this problem by a procedure based on cofactoring and theory simplification. Although our approach may cause a blow-up, it often simplifies the formula in practice. Our approach is applied to linear arithmetic logic in this paper; however, it can be easily applied to other logics like the logic of equality and uninterpreted function symbols (EUF), the logic of bit-vector, or the logic of arrays. Only the terminal cases are different in each logic. Our experiments show that our approach is promising and often speeds up a solver by orders of magnitude. The experiments also demonstrate the effectiveness of theory simplification.

The rest of this paper is organized as follows. Section 2 defines notation and summarizes the main concepts. Section 3 discusses motivation and outlines our approach to the problem. Section 4 presents the simplifications applied before invoking the term-ITE conversion. Section 5 presents an algorithm for term-ITE conversion with theory reasoning. After a survey of related work in Sect. 6, experiments are presented in Sect. 7, and conclusions are offered in Sect. 8.

2 Preliminaries

We consider the satisfiability problem for linear arithmetic logic, which is the quantifier-free fragment of first-order logic that deals with linear arithmetic constraints. Let V_B be a set of propositional variables and V_R be a set of real-valued variables. The formulae in linear arithmetic logic are inductively defined as the largest set that satisfies the following rules.

- A propositional variable $a \in V_B$ is a formula.
- A real number $c \in \mathbb{R}$ is a (constant) term.
- The product cx of a real number $c \in \mathbb{R}$ and a real-valued variable $x \in V_R$ is a term.
- If t_1 and t_2 are terms, then $t_1 + t_2$ is a term.
- If t_1 and t_2 are terms, and f is a formula, then *term-ite*(f, t_1, t_2) is a term.
- If t_1 and t_2 are terms, and $\sim$ is a relational operator in $\{=, \neq, <, \leq, >, \geq\}$, then $t_1 \sim t_2$ is a formula.
- If f_1, f_2, and f_3 are formulae, then $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ and *ite*(f_1, f_2, f_3) are formulae.

The semantics are defined in the usual way; in particular, *ite*(f_1, f_2, f_3) is equivalent to $(f_1 \wedge f_2) \vee (\neg f_1 \wedge f_3)$. An *atomic formula* is one of the form $t_1 \sim t_2$. A *positive literal* is an atomic formula or a propositional variable; a *negative literal* is the negation of a positive literal.

A model for a formula f is an assignment of values to the variables in the formula that is consistent with the type of each variable and that makes the formula true. A formula that has at least one model is *satisfiable*. In recent years, decision procedure for *LA*, and other fragments of quantifier-free first-order logic, have been based on the DPLL procedure. formula $\mathcal{F}$, a propositional abstraction $\mathcal{F}_b$ of $\mathcal{F}$ is built by substituting each atomic formula with a new propositional variable. As the DPLL procedure provides a model for $\mathcal{F}_b$, a *theory solver* for *LA* is invoked with the set of atomic formulae that are assigned. The theory solver checks the feasibility of the set. If the set is feasible, then the model is also a model in theory. If the set is infeasible, then the explanation of the infeasibility is returned to the DPLL procedure. The procedure continues until it finds a complete model, or decides that $\mathcal{F}$ is *unsatisfiable* in the given theory.

3 Term-ITE Conversion

An *LA* formula can often be expressed more concisely by using term-ITEs. For example, Fig. 2 shows that the formula f in (a) is equivalent to the more verbose formula f' in (b). Despite the conciseness afforded by term-ITEs, a *LA* formula with term-ITEs is often converted into a formula without them, so that the formula may be solved by an SMT solver based on the propositional abstraction.

3.1 Two Methods for Term-ITE Conversion

A common way to eliminate these term-ITEs is to introduce a fresh constant that replaces the term-ITE. In particular, an *LA* formula $f($*term-ite*$(g, t_1, t_2))$ is converted to the equisatisfiable

$$f(c) \wedge \textit{ite}(g, t_1 = c, t_2 = c) \ , \tag{1}$$

where c is a constant that does not appear in the given formula. The advantage of this conversion is that it does not blow up; however, it often retains redundancies in the converted formula. For example, the formula *term-ite*$(g, 1, 2)$ = *term-ite*$(h, 3, 4)$ can be reduced to $\bot$, whereas the conversion generates *ite*$(g, c = 1, c = 2) \wedge$ *ite*$(h, c = 3, c = 4)$ that contains a redundancy. To remove the redundancy, additional theory

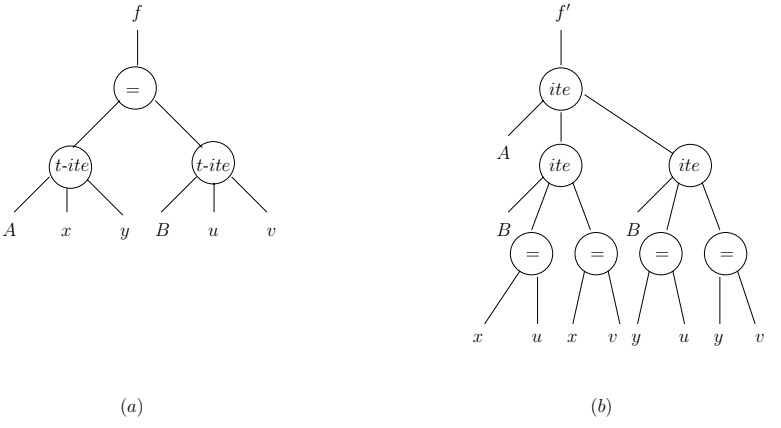


Fig. 2. Term-ITE conversion

reasoning is required. A naive approach to the term-ITE conversion will be to combine every term in the left-hand side of the relational operator with the terms in the right-hand side depending on the conditional terms of term-ITEs. In particular, an *LA* formula $f(\textit{term-ite}(g, t_1, t_2))$ is converted according to following conversion rule [7].

$$f(\textit{term-ite}(g, t_1, t_2)) \iff \textit{ite}(g, f(t_1), f(t_2)) \ . \tag{2}$$

This approach removes the redundancy in the above example on the fly; however, as Fig. 2 illustrates, the converted formula may grow exponentially large in the worst case.

3.2 Term-ITE Conversion with Cofactors

As an alternative to the approaches described in Sect. 3.1, term-ITE conversion can be done by computing cofactors.

Definition 1. *Let $f(x_1, ..., x_n)$ be an LA formula, where each x_i is a positive literal. Then,*

$$f_{x_i} = f(x_1, ..., x_{i-1}, \top, x_{i+1}, ..., x_n)$$
$$f_{\neg x_i} = f(x_1, ..., x_{i-1}, \bot, x_{i+1}, ..., x_n)$$

are the positive and negative cofactors of f with respect to x_i.

Theorem 1 (Boole). *Let $f(x_1, ..., x_n)$ be an LA formula. Then $f(x_1, ..., x_n) = (x_i \wedge f_{x_i}) \vee (\neg x_i \wedge f_{\neg x_i}) = \textit{ite}(x_i, f_{x_i}, f_{\neg x_i})$.*

According to Theorem 1, the following rule can be used to rewrite an *LA* formula:

$$f(\textit{term-ite}(g, t_1, t_2)) \iff \textit{ite}(x, f_x(\textit{term-ite}(g, t_1, t_2)), f_{\neg x}(\textit{term-ite}(g, t_1, t_2))) \ . \tag{3}$$

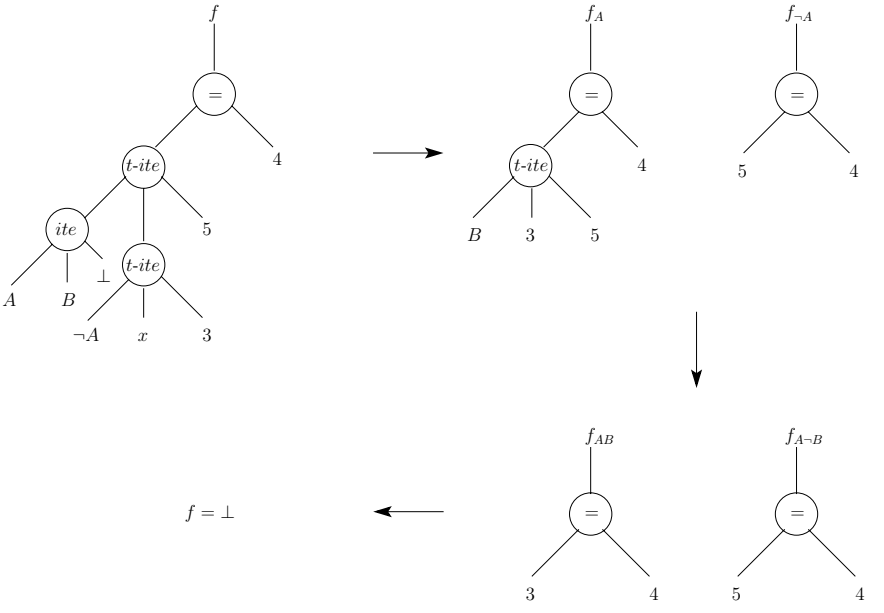


Fig. 3. Term-ITE conversion with cofactor

By computing the cofactors of f, the conversion may greatly simplify the converted formula. In Fig. 3, f is simplified to $\bot$ using (3). In particular, the cofactors $f_A \iff (\textit{term-ite}(B, 3, 5) = 4)$ and $f_{\neg A} \iff (5 = 4) \iff \bot$ are first computed. Then f is simplified to $(A \wedge f_A)$, and finally reduced to $\bot$ by cofactoring f_A with respect to B.

This kind of simplification can often be applied to the *LA* problems in SMT-LIB [14]. As the previous example shows, the simplification for equality is easily done by comparing two constants. On the other hand, if fresh constants are introduced, redundancy may remain in the converted formula: a fresh constant c replaces the term $\textit{term-ite}(\textit{ite}(A, B, \bot), \textit{term-ite}(\neg A, x, 3), 5)$ in f. Then f is rewritten in two steps: first as

$$(c = 4) \wedge \textit{ite}(\textit{ite}(A, B, \bot), c = \textit{term-ite}(\neg A, x, 3), c = 5) \ ,$$

and then as

$$(c = 4) \wedge (c' = c) \wedge \textit{ite}(\textit{ite}(A, B, \bot), \textit{ite}(\neg A, c' = x, c' = 3), c = 5) \ ,$$

where c' is another fresh constant. Removing the redundancy from the converted formula requires theory reasoning. While such reasoning is uncomplicated in this example, in general the new constants may make it cumbersome. Although the cofactoring method may give a huge reduction, it may blow up if there is little simplification. Compared to the approach that introduces a fresh constant, it is more aggressive.

Definition 2. *Let x be a literal and h be a formula. We write $x \models_T h$ if h is a consequence of x in theory T, and we call h a* theory consequence *of x.*

The cofactoring method can be further extended with theory reasoning. Using the theory propagation method [12], an assignment to an atomic predicate may entail assignments to other atomic predicates. For example, in *LA*, if we make an assignment to $(x < 0) = \top$, then $(x < 3) = \top$ and $(x > 1) = \bot$. The following rules show how theory propagation may help in the simplification of the converted formula:

$$\frac{x \models_T h}{f_x(\textit{term-ite}(h, t_1, t_2)) \iff f_x(t_1)} \tag{4}$$

$$\frac{x \models_T \neg h}{f_x(\textit{term-ite}(h, t_1, t_2)) \iff f_x(t_2)} \,. \tag{5}$$

As we compute the cofactors in the term-ITE conversion, we make an assignment to the cofactoring literal. If the cofactoring literal is an atomic formula and the computed cofactor is also an atomic formula, then theory reasoning can be invoked to check the relation between these two atoms. The following consequence of Theorem 1 gives an idea of how this simplification can be done; it will be used in Sect. 5.

Theorem 2. *Given a formula f of theory T and a literal x_i, if $x_i \models_T f_{x_i}$, then $f \iff x_i \vee f_{\neg x_i}$. If $x_i \models_T \neg f_{x_i}$, then $f \iff \neg x_i \wedge f_{\neg x_i}$.*

4 Simple Preprocessing

Before we execute term-ITE conversion for an *LA* formula f, terminal cases for term-ITE are detected and basic simplification is carried out. Let $a \in V_B$; let t_1, t_2, and t_3 be terms and let c_1, c_2, and c_3 be constants. In the *LA* formula, we detect special cases like *term-ite*$(\top, t_1, t_2) \iff t_1$, *term-ite*$(\bot, t_1, t_2) \iff t_2$, *term-ite*$(a, t_1, t_1) \iff t_1$. We also simplify nested term-ITEs such as *term-ite*$(a,$ *term-ite*$(a, t_1, t_3), t_2) \iff$ *term-ite*(a, t_1, t_2), *term-ite*$(a,$ *term-ite*$(\neg a, t_3, t_2), t_1) \iff$ *term-ite*(a, t_2, t_1). For arithmetic terms, $(0 + t_1) \iff t_1, (0 \cdot t_1) \iff 0, (1 \cdot t_1) \iff t_1, (-(-t_1)) \iff t_1, (c_1 + c_2) \iff c_3$, where c_3 is the sum of c_1 and c_2.

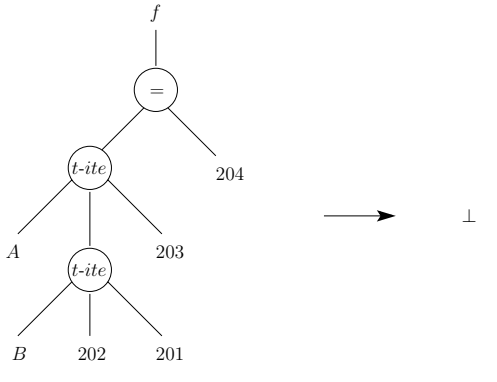


Fig. 4. Term-ITE conversion with simple check

Furthermore, if a formula f has a root node that is a relational operator applied to term-ITEs and has leaves that are all constants, then it can be simplified. For simplicity, we only check the case where either of the children of the root node is a constant. Example 1 shows such a case.

Example 1. Let f be a formula shown in Fig. 4. The formula f is an equality with term-ITEs. As Fig. 4 shows, the terms on the left-hand side of the root node are all constants and the one on the right-hand side is also a constant. In such a case, we compare all the constants in the left hand side for equality with the constant on the right, 204. Clearly, $(202 = 204) \iff \bot$, $(201 = 204) \iff \bot$ and $(201 = 203) \iff \bot$; hence $f = \bot$.

5 Algorithm

We assume that an SMT solver adopts the rewriting procedure. Given an *LA* formula $\mathcal{F}$ with term-ITEs, an SMT solver converts $\mathcal{F}$ into $\mathcal{F}'$ by removing all term-ITEs in $\mathcal{F}$. The SMT solver then decides the satisfiability of $\mathcal{F}'$. In this section, we describe how $\mathcal{F}$ is converted into $\mathcal{F}'$.

As the pseudocode of Fig. 5 shows, the main function of term-ITE conversion is called with an *LA* formula $\mathcal{F}$. The formula $\mathcal{F}$ is represented as a directed acyclic graph (DAG), where each node is a Boolean operator, a relational operator, an arithmetic operator, a term-ITE, or an atom. The conversion is applied to each relational operator in the DAG, and the procedure ends when $\mathcal{F}'$ no longer contains term-ITEs. The main function starts by selecting the candidates for the conversion in the DAG. Each candidate is a relational operator that has a term-ITE as a descendant, and the candidates are gathered in F. As Line 4 in Fig. 5 shows, the term-ITE conversion is invoked with $f \in F$, and all the term-ITEs are removed from f. After the conversion of f, the converted formula f' is either a Boolean ITE or an atom. The procedure ends when all $f \in F$ have been considered. At that point, $\mathcal{F}$ has been converted into $\mathcal{F}'$, which does not contain any term-ITEs.

As TermIteConversion is invoked with $f \in F$, a cofactoring variable v is searched for in f at Line 10. We select an atom as a cofactoring variable that resides in the conditional term of the term-ITE. With v, we recursively compute the cofactor of f. In general, the cofactors are computed for the children of f with respect to v, and a new formula f_v is created with new children. As shown in Line 38 of Fig. 6, if f is a relational operator, we compute the cofactors l_v and r_v for the children of f. After computing the cofactors, we check for simple cases with l_v and r_v. The simple check detects terminal cases for the terms l_v and r_v with respect to the type $(=, <, \leq, >, \geq)$ of f. Figure 4 shows an example of simplification. If a terminal case is not found, a new formula f_v is generated with $\mathrm{type}(f)$, l_v and r_v. The newly generated formula, f_v is either an atom or a relation operator with term-ITEs. In the latter case, term-ITE conversion is called with f_v, again. In Line 47 of Fig. 6, if f_v is an atom, theory reasoning is done with v. As Theorem 2 shows, if $v \models_T f_v$, then f in Line 13 of Fig. 5 is simplified to $v \vee f_{\neg v}$. Likewise, if $v \models_T \neg f_v$, then f is simplified to $\neg v \wedge f_{\neg v}$. When f is either a term-ITE or a Boolean ITE, the cofactor for each term of f is computed as shown in Line 58 of Fig. 6. As in the cofactoring on the relational operator, a terminal

case is checked for the conditional term f_c. If f_c is an atomic predicate, theory reasoning is done with v and f_c using Rules 4–5 of Sect. 3.2. If a terminal case is not found, then the cofactors for the terms of f are computed to obtain f_v.

Example 2. If f is a relational operator such that $D(f)$ contains term-ITEs, we convert f into f' such that there is no term-ITE in $D(f')$. In Fig. 7, let $A \leftrightarrow (x \geq 50)$ and $B \leftrightarrow (y \leq 58)$. We first traverse $D(f)$ to find a cofactoring variable. We pick an atomic

```
1  TermIteConversionMain (F) {
2      F := GatherCandidateForTermIteConversion (F);
3      for (each f ∈ F in topological order) {
4          f' := TermIteConversion (f);
5          F' := UpdateFormula (F, f');
6      }
7      return F';
8  }

9  TermIteConversion (f) {
10     while ( v := GetCofactorVariable (f) ) {
11         f_v := CofactorRecur (f, v);
12         f_¬v := CofactorRecur (f, ¬v);
13         f := Ite (v, f_v, f_¬v);
14     }
15     return f;
16 }

17 CofactorRecur (f, v) {
18     if ( f = v ) {
19         f_v := ⊤;
20     } else if ( f = ¬v ) {
21         f_v := ⊥;
22     } else if ( is_relation(f) ) {
23         f_v := CofactorRelRecur (f, v);
24     } else if ( is_term_ite(f) ) {
25         f_v := CofactorTiteRecur (f, v);
26     } else { /* +, −, × */
27         C := children(f);
28         For each c ∈ C {
29             d := CofactorRecur (c, v);
30             Add(D, d);
31         }
32         f_v := NewFormula (type(f), D); /* type(f) is either +, −, ×. */
33         SimplifyArithFormula(f_v);
34     }
35     return f_v;
36 }
```

Fig. 5. Term-ITE conversion algorithm

```
37 CofactorRelRecur (f, v) {
38     l_v := CofactorRelRecur (left(f), v);
39     r_v := CofactorRelRecur (right(f), v);
40     f_v := SimpleCheckWithTerms (type(f), l_v, r_v);
41     if ( f_v = NoSimplification ) {
42         f_v := NewFormula (type(f), l_v, r_v);
43         if ( is_term_ite(l_v) or is_term_ite(r_v) ) {
44             f_v = TermIteConversion (f_v);
45         }
46     }
47     if ( is_atom(f_v) ) {
48         if ( v ⊨_T f_v ) { /* theory reasoning */
49             f_v := ⊤
50         } else if ( v ⊨_T ¬f_v ) { /* theory reasoning */
51             f_v := ⊥
52         }
53     }
54     return f_v;
55 }

56 CofactorTiteRecur (f, v) {
57     f_c := CondTerm(f); f_t := ThenTerm(f); f_e := ElseTerm(f);
58     if ( f_c = ⊤ ) {
59         return CofactorRecur (f_t, v);
60     } else if ( f_c = ⊥ ) {
61         return CofactorRecur (f_e, v);
62     } else if ( is_pred(f_c) ) {
63         if ( v ⊨_T f_c ) { /* theory reasoning */
64             return CofactorRecur (f_t, v);
65         } else if ( v ⊨_T ¬f_c ) ) { /* theory reasoning */
66             return CofactorRecur (f_e, v);
67         }
68     }
69     c_v := CofactorRecur (f_c, v);
70     t_v := CofactorRecur (f_t, v);
71     e_v := CofactorRecur (f_e, v);
72     f_v := Ite (c_v, t_v, e_v);
73     return f_v;
74 }
```

Fig. 6. Term-ITE conversion algorithm

formula A as cofactoring variable and compute the cofactors of f with respect to A. As we proceed, $f_A = (36 \leq 55) = \top$ and $f_{\neg A}$ is constructed with a new term-ITE. Since there still exists a term-ITE in $D(f_{\neg A})$, we look for another cofactoring variable in $f_{\neg A}$. We select B and compute the cofactors for $f_{\neg A}$. As a result, we get $f_{\neg AB} = (x \leq 55)$ and $f_{\neg A \neg B} = (y \leq 55)$. Since $A \models_T f_{\neg AB}$ and $\neg B \models_T \neg f_{\neg A \neg B}$, $f_{\neg AB} = \top$ and $f_{\neg A \neg B} = \bot$. Finally, the converted formula f' gets reduced to $ite(A, \top, B)$ as shown in Fig. 7.

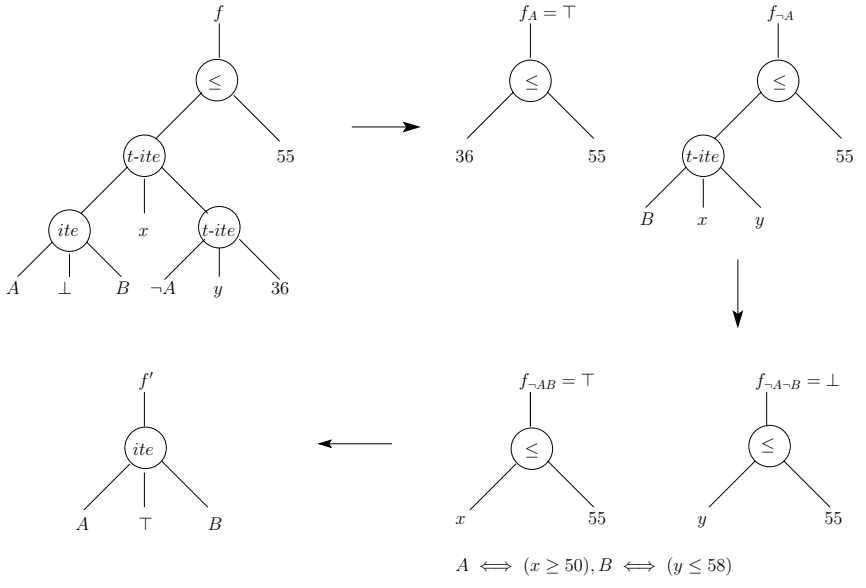


Fig. 7. Term-ITE conversion

6 Related Work

Early references on the treatment of ITEs are [8], [2] and [7]. For SMT preprocessing, HTP [13] introduces several preprocessing techniques such as unate predicate detection, variable substitution and symmetry breaking. Yices [3] uses a Gaussian elimination to reduce the size of initial tableau of equality constraints. In [17], Yu *et al.* describes a static learning technique that analyzes the relationship of the linear constraints. In Karplus's technical report [8], a new canonical form for *ITE* DAGs is introduced using two-cuts, and *ITE* normalization using recursive transformation is shown in [11].

7 Experimental Results

We have implemented the algorithm presented in Sect. 5 in Sateen [10, 9, 15], a theorem prover for quantifier-free first-order logic that combines the propositional reasoning engine of [5, 6] with theory-specific procedures. Experiments are done with the full set of QF_LIA (Quantifier free linear integer arithmetic logic) benchmarks from SMT-COMP (Satisfiability Modulo Theories Competition) [14]. The experiments were performed on an Intel 2.4 GHz Quad Core with 4 GB of RAM running Linux. Time out was set at 1000 seconds. Sateen was compared with Z3.2 [14], MathSAT-4.2[1, 14] and Yices-1.0.16 [16]. Z3.2 and MathSAT-4.2 are the ones that were submitted to SMT-COMP in 2008. We used most recent version of Yices that is available.

In QF_LIA benchmarks, there are two benchmark sets, *nec-smt* and *rings*, that are rich in term-ITE operators. More than 90 percent of the QF_LIA benchmarks belong to those two sets. The instances in the *nec-smt* set are generated by the SMT-based BMC

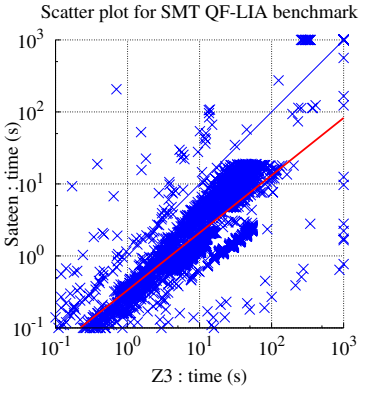


Fig. 8. Z3 vs. Sateen on QF_LIA

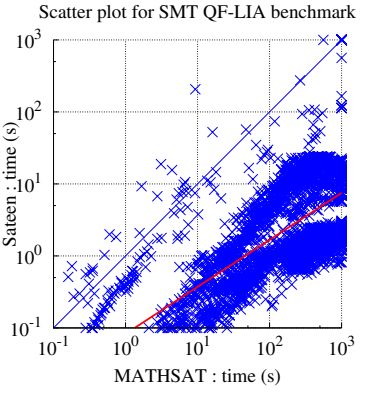


Fig. 9. MATHSAT vs. Sateen on QF_LIA

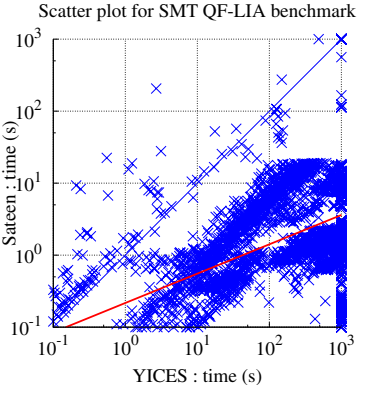


Fig. 10. YICES vs. Sateen on QF_LIA

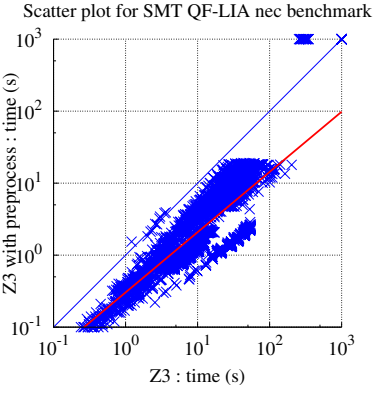


Fig. 11. Z3 WITH PREPROCESS vs. Z3 on QF_LIA

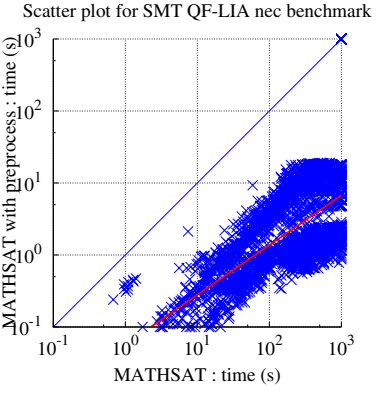


Fig. 12. MATHSAT WITH PREPROCESS vs. MATHSAT on QF_LIA

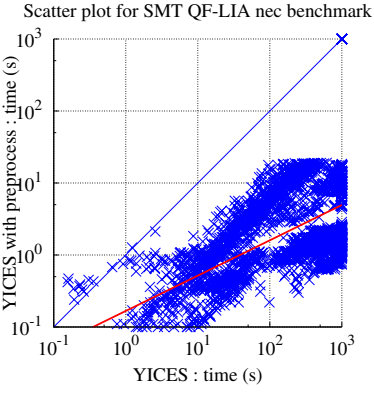


Fig. 13. YICES WITH PREPROCESS vs. YICES on QF_LIA

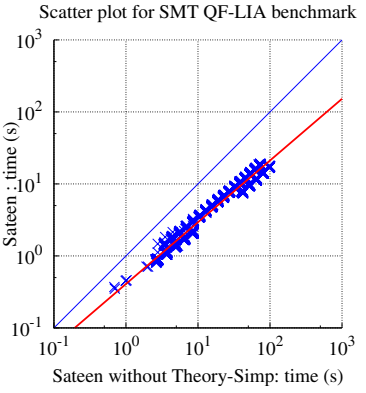


Fig. 14. SATEEN vs. Sateen without Theory-Simp on QF_LIA

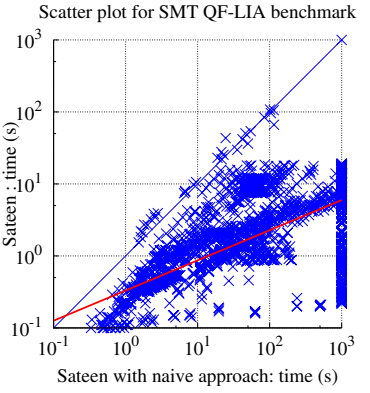


Fig. 15. SATEEN vs. Sateen with naive approach on QF_LIA

engine of F-Soft [4]; the instances in *rings* encode associativity properties on modular arithmetic.

Figures 8–10 show scatterplots comparing Z3, MathSAT and Yices to Sateen. Points below the diagonal represent wins for Sateen. Each scatterplot shows two lines: The main diagonal, and $y = \kappa \cdot x^{\eta}$, where κ and η are obtained by least-square fitting. Figure 8 shows that Sateen is often an order of magnitude faster than Z3. In Fig. 9 and 10, Sateen is often a few orders of magnitude faster than MathSAT and Yices.

We further evaluated our preprocessor by generating simplified formulae from the *nec-smt* benchmarks and running Z3, MathSAT, and Yices on them. All solvers took less than a second on each simplified problem. Figures 11–13 show scatterplots comparing Z3, MathSAT and Yices with preprocessor and without preprocessor. The times for the solvers with preprocessor include preprocessing time. As Figures 11–13 show, our preprocessor is also effective for other solvers.

Table 1 shows the number of term-ITE reductions with the simple preprocessing on randomly selected benchmarks. The first column gives the name of the benchmarks, the

Table 1. Number of term-ITE reduction with simple preprocessing

Benchmark	Before S.P.	After S.P.	rate(%)
bftpd_login/prp-74-50.smt	38773	34085	12
checkpass/prp-10-46.smt	17240	14949	13
checkpass/prp-63-50.smt	25376	21893	14
checkpass_pwd/prp-38-42.smt	12196	10354	15
getoption/prp-2-200.smt	11269	9791	13
getoption_directories/prp-0-110.smt	72892	62457	14
getoption_group/prp-72-49.smt	15021	12094	20
handler_sigchld/prp-20-46.smt	7800	6824	13
int_from_list/prp-34-41.smt	7184	5888	18
user_is_in_group/prp-23-48.smt	22549	17939	20

second one is the initial number of term-ITEs, and the third one is the number of term-ITEs after the simple preprocessing. The last column gives the rate of the reduction. On average, we achieved 15% term-ITE reduction with the simple preprocessing of Section 4.

Finally, we compared our approach to the naive approach of Eq. 2. As Fig. 15 shows, our approach is significantly better. In addition, we disabled theory simplification in the algorithm and ran the experiment on the problems where the simplifications play a significant role. Figure 14 shows that Sateen with theory simplification is consistently better than the one without simplification.

8 Conclusions

We have presented an algorithm for the term-ITE conversion in first-order theories like the theory of linear arithmetic. The approach is based on the computation of cofactors and theory simplification. The simplification is done by detecting special cases in the formula or using theory propagation on the atomic predicates. Experiments show that this approach is very effective in most QF_LIA benchmarks and often speeds up SMT solvers. On the other hand, since our approach may still blow up in general, we are working on combining it with a less aggressive approach, based on (1), that does not blow up.

Acknowledgment. The authors thank the reviewers for their detailed suggestions.

References

[1] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., Sebastiani, R.: An incremental and layered procedure for the satisfiability of linear arithmetic logic. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 317–333. Springer, Heidelberg (2005)

[2] Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: Proceedings of the 27th Design Automation Conference, Orlando, FL, June 1990, pp. 40–45 (1990)

[3] Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)

[4] Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-soft: Software verification platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005)

[5] Jin, H., Han, H., Somenzi, F.: Efficient conflict analysis for finding all satisfying assignments of a Boolean circuit. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 287–300. Springer, Heidelberg (2005)

[6] Jin, H., Somenzi, F.: Prime clauses for fast enumeration of satisfying assignments to Boolean circuits. In: Proceedings of the Design Automation Conference, Anaheim, CA, June 2005, pp. 750–753 (2005)

[7] Jones, R.B., Dill, D.L., Burch, J.R.: Efficient validity checking for processor verification. In: Proceedings of the International Conference on Computer-Aided Design, San Jose, CA, November 1995, pp. 2–6 (1995)

[8] Karplus, K.: Representing Boolean functions with if-then-else DAGs. In Technical Report UCSC-CRL-88-28, Board of Studies in Computer Engineering, University of California at Santa Cruz, Santa Cruz, CA 95064 (December 1988)
[9] Kim, H., Jin, H., Ravi, K., Spacek, P., Pierce, J., Kurshan, B., Somenzi, F.: Application of formal word-level analysis to constrained random simulation. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 487–490. Springer, Heidelberg (2008)
[10] Kim, H., Jin, H., Somenzi, F.: Disequality management in integer difference logic via finite instantiations. Journal on Satisfiability, Boolean Modeling and Computation 3, 47–66 (2007)
[11] Nelson, G., Oppen, D.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (2008)
[12] Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
[13] Roe, K.: The heuristic theorem prover: Yet another SMT modulo theorem prover. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 467–470. Springer, Heidelberg (2006)
[14] http://smtcomp.org/
[15] http://vlsi.colorado.edu/~vis
[16] http://yices.csl.sri.com
[17] Yu, Y., Malik, S.: Lemma learning in SMT on linear constraints. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 142–155. Springer, Heidelberg (2006)

On-the-Fly Clause Improvement*

Hyojung Han and Fabio Somenzi

University of Colorado at Boulder
{Hhhan,Fabio}@Colorado.EDU

Abstract. Most current propositional SAT solvers apply resolution at various stages to derive new clauses or simplify existing ones. The former happens during conflict analysis, while the latter is usually done during preprocessing. We show how subsumption of the operands by the resolvent can be inexpensively detected during resolution; we then show how this detection is used to improve three stages of the SAT solver: variable elimination, clause distillation, and conflict analysis. The "on-the-fly" subsumption check is easily integrated in a SAT solver. In particular, it is compatible with the strong conflict analysis and the generation of unsatisfiability proofs. Experiments show the effectiveness of this technique and illustrate an interesting synergy between preprocessing and the DPLL procedure.

1 Introduction

In the last decade, advances in the satisfiability checking (SAT) of propositional formulae have been achieved through a variety of techniques to efficiently prune the search space and algorithms to improve the quality of the input formula.

Simplifying the CNF clauses speeds up Boolean Constraint Propagation (BCP) and accelerates detection of conflicts. Techniques like subsumption, variable elimination, and distillation have had significant success in practice. Preprocessing may solve some simple problems by itself, but usually does not remove all redundant clauses. That is because it has to trade off the reduction in the input formula against to the time spent.

Clause recording adds *conflict-learned clauses* or, simply, *conflict clauses* to the original SAT instance. Each conflicting assignment is analyzed to identify a subset that is sufficient to cause the current conflict. The disjunction of the literals in the subset becomes a new clause added to the original SAT instance.

Previous work has addressed the quality of conflict clauses [7, 13, 11, 6]. In particular, strong conflict analysis proposed in [6] generates a second conflict clause that is often more effective than a regular conflict clause of [13] in escaping regions of the search space where the solver would otherwise linger for long time. A common thread of most work on the subject is the search for a balance between a technique's cost and its ability of to detect implications—the *deductive power* of [4].

In this paper we propose an algorithm that detects subsumptions during resolution during both preprocessing and conflict analysis with the minimal extra effort required to compare the lengths of operands and resolvent. Our on-the-fly subsumption check can be easily applied to both strong and regular conflict analysis. We show how this

* This work was supported in part by SRC contract 2008-TJ-1859.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 209–222, 2009.

inexpensive check is used to improve deductive power at three stages of the SAT solver: variable elimination, clause distillation, and conflict analysis.

Experiments show that the on-the-fly subsumption check proposed produces a great effect on run time of both the SAT solver and the preprocessing stage. Moreover, the results illustrate an interesting synergy between preprocessing techniques and on-the-fly subsumption check in the DPLL procedure.

Related to on-the-fly simplification is the problem of finding compact conflict clauses after conflict analysis. *Conflict clause minimization* [2] tests every literal in a newly-generated conflict clause. It removes any literal in the conflict clause if its negation is implied by the other literals in the clause. To do this, it also uses a resolution-based subsumption check (self subsumption) applied to the conflict clause and the antecedent clause. However, in contrast to on-the-fly simplification, this method does not simplify existing clauses. The minimization algorithm traverses only the part beyond the UIP in the implication graph, while our proposed simplification algorithm traverses between the conflicting clause and the first UIP. *Assignment shrinking* [7] is a technique to derive a new conflict clause, which tends to be more compact than the conflict clause. The algorithm of [7], after generating a conflict learned clause, backtracks to the highest level to undo all the assignments in the conflict clause. It then starts replicating the assignments to the literals involved in the previous conflict, until a new conflict occurs. This may produce a new smaller conflict clause. Since this is an expensive technique, its invocation is controlled by a criterion based on the length of the conflict clause.

An existing clause may be subsumed by a conflict clause newly found by any of the conflict analysis algorithms. Hence, one may try to simplify the newly redundant clauses. On-the-fly simplification algorithm used in [12] can detect the subsumed clause with a *one watched literal* scheme, when a new clause is generated by conflict analysis. In spite of the efficiency afforded by the one watched literal scheme, checking the subsumption relation between clauses requires significant extra work.

The rest of this paper is organized as follows. Background material is covered in Section 2. In Section 3 we describe the principles of our on-the-fly clause improvement approach during conflict analysis and present the details of the algorithm. Section 4 discusses how the subsumption check applies to the preprocessing stages, i.e., variable elimination and distillation of clauses. Section 5 reports results from the implementation of the proposed approach. We draw conclusions and outline future work in Sect. 6.

2 Preliminaries

In this paper we assume that the input to the SAT solver is a formula in Conjunctive Normal Form (CNF). A CNF formula is a set of *clauses*; each clause is a set of *literals*; each literal is either a variable or its negation. The function of a clause is the disjunction of its literals, and the function of a CNF formula is the conjunction of its clauses.

$$\{\{\neg a, c\}, \{\neg b, c\}, \{\neg a, \neg c, d\}, \{\neg b, \neg c, \neg d\}\}$$

corresponds to the following propositional formula:

$$(\neg a \vee c) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg c \vee d) \wedge (\neg b \vee \neg c \vee \neg d).$$

Clause γ_1 subsumes clause γ_2 if $\gamma_1 \subseteq \gamma_2$. Given $\gamma_1 = \gamma_1' \cup \{l\}$ and $\gamma_2 = \gamma_2' \cup \{\neg l\}$, the *resolvent* of γ_1 and γ_2 is $\gamma_1' \cup \gamma_2'$ and is implied by $\{\gamma_1, \gamma_2\}$. An *assignment* for CNF formula F over the set of variables V is a mapping from V to $\{\mathsf{true}, \mathsf{false}\}$. A *partial* assignment maps a subset of V. A satisfying assignment for CNF formula F is one that causes F to evaluate to true. We represent assignments by sets of *unit* clauses, that is, clauses containing exactly one literal. For instance, the partial assignment that sets a and b to true and d to false is written $\{\{a\}, \{b\}, \{\neg d\}\}$ or, interchangeably, $a \wedge b \wedge \neg d$. A literals assigned under the partial assignment may be annotated with a *decision level*, the number of following the @ sign. For example, a assigned true at level 1 is written $a@1$. A clause γ is *asserting* under assignment A if all its literals except one (the asserted literal) are false. We say that an asserting clause is an *antecedent* of its asserted literal.

Many successful SAT solvers are based on the DPLL procedure, whose modern incarnations are described by the pseudocode of Fig. 1. The solver maintains a current partial assignment that is extended until it either becomes a total satisfying assignment, or becomes conflicting. In the latter case, the solver analyzes the conflict and *backtracks* accordingly. Conflict analysis [10]leads to learning a *conflict clause*, that is, a clause computed from repeated applications of resolution to the conflicting clause and the *antecedent clause* of the literal in the conflicting clause that was most recently implied.

```
1  GRASP_DPLL() {
2    while (CHOOSENEXTASSIGNMENT() == FOUND)
3      while (DEDUCE() == CONFLICT) {
4        blevel = ANALYZECONFLICT();
5        if (blevel < 0) return UNSATISFIABLE;
6        else BACKTRACK(blevel);
7      }
8    return SATISFIABLE;
9  }
```

Fig. 1. GRASP_DPLL algorithm

The GRASP_DPLL procedure is often applied after a preprocessing phase, which attempts to remove redundant clauses and literals to speed up SAT solvers. SatELite [1, 9] simplifies a CNF formula by iteratively checking the subsumption relation between clauses and eliminates variables by resolution. For instance, $a \vee b$ and $a \vee \neg b \vee c$ give a resolvent $a \vee c$ that subsumes the latter; this is called *self subsumption*. Resolution can also be applied to eliminate variables from the formula. If, for example, $a \vee b$, $\neg b \vee c$ and $\neg b \vee d$ are the only clauses containing b, then they can be replaced by $a \vee c$ and $a \vee d$ while guaranteeing *equisatisfiability*. Since the elimination of variables may increase the number of clauses, it is used in a limited way in preprocessing. In [4], the CNF formula is *distilled* to increase its *deductive power*. The transformation is done by asserting the negation of each in a clause until either a conflict is found, or one of the literals of the clause is implied true. In both cases, the distillation procedure analyzes the implication graph to generate the improved clause.

Example 1. Consider the following formula:

$$(a \vee b \vee \neg c) \wedge (a \vee c \vee d) \wedge (b \vee c \vee e) \wedge (\neg d \vee f) \wedge (\neg e \vee g) \wedge (\neg f \vee \neg g \vee h) \wedge (\neg f \vee \neg g \vee \neg h)$$

Suppose that the decisions $\{\neg a@1, \neg b@2\}$ are made by the SAT solver and that the implications of those decisions are computed. Figure 2 shows the implication graph that represents the implications derived up to the current decision level. Directed edges in the graph are labeled with clause numbers. The implications result in a conflict on variable h, that is, two opposite assignment to h at the same time. Conflict analysis is therefore invoked. The implication graph in Fig. 2 also shows each resolvent γ_i that the conflict analysis generates while traversing backward the implication graph from the conflicting clause c_7. □

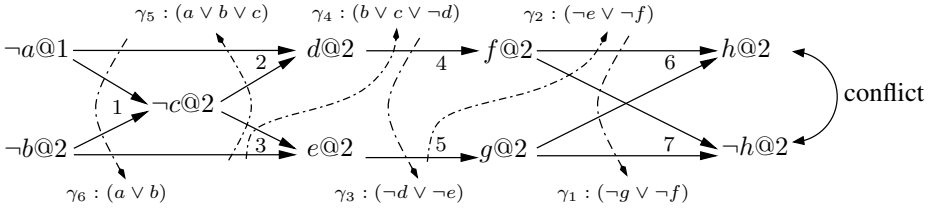


Fig. 2. Implication graph for the first conflict of Example 1

Most conflict analysis algorithms terminate as soon as they find a clause containing a *Unique Implication Point* (UIP), that is, a single assignment made at the current level. For this case, since γ_6 contains the first UIP, that is literal b, it is chosen as conflict clause. However, the UIP is in this case the decision variable. When the first UIP is far from the conflict in the implication graph, the conflict clause may not be effective in preventing the SAT solver from repeating the same mistake. *Strong conflict analysis* [6] can be a remedy in such cases: It examines intermediate resolvents as UIP-based conflict analysis does. Contrary to UIP-based analysis, however, it generates an additional conflict clause that contains more than one literal assigned at the current decision level. This additional conflict clause must be one of the intermediate resolvents derived between the conflict and the first UIP. Usually, the closer to the conflict, the fewer literals the resolvent contains. Therefore, the additional conflict clause tends to be shorter than the conflict clause with 1-UIP.

3 On-the-Fly Simplification Based on Resolution

Detecting whether the resolvent of two clauses subsumes either operand is easy and inexpensive. Therefore, checking *on-the-fly* for subsumption can be added with almost no penalty to those operations of SAT solvers that are based on resolution. In this section we review the basic idea and detail its application to conflict analysis. Later, we discuss on-the-fly subsumption in preprocessing.

An efficient on-the-fly check for subsumption during resolution is based on the following elementary fact.

Lemma 1. *Let* $c_1 = c_1' \cup \{l\}$ *and* $c_2 = c_2' \cup \{\neg l\}$ *be two clauses. Their* resolvent $c_1' \cup c_2'$ *subsumes* c_1 (c_2) *iff* $|c_1' \cup c_2'| = |c_1| - 1$ ($|c_1' \cup c_2'| = |c_2| - 1$).

Thanks to Lemma 1, it is possible to detect existing clauses that are subsumed by resolvents and replace them with the resolvents themselves. Doing so during conflict analysis is easy because the eliminated literal is the one asserted by the clause itself. If that literal is kept in first position in the clause [3], it is easily accessed. In variable elimination, as we shall see, the literal to be removed corresponds to the variable that is being eliminated. Therefore, it is enough to save its position while scanning a clause. In summary, the overhead of on-the-fly subsumption check is negligible. The advantages, on the other hand, may be significant as illustrated by the following example.

Example 2. Consider the following set of clauses:

$$F = (a \vee b \vee \neg c) \wedge (a \vee b \vee \neg d) \wedge (c \vee d \vee \neg e) \wedge (c \vee e \vee f) \wedge (d \vee e \vee \neg f) \wedge (\neg b \vee \neg d \vee e) \wedge (\neg d \vee \neg e)$$

Suppose that the first decision is to set a to false, and the second decision is to set b to false. From these decisions literals $\neg c$, $\neg d$, and $\neg e$ are deduced at level 2. This partial assignment, in turn, yields f and $\neg f$ through the fourth and the fifth clause. Analysis of this conflict produces the implication graph shown in Fig. 3.

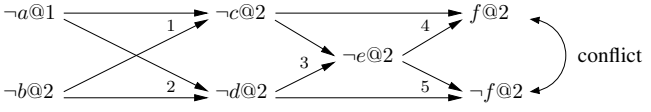


Fig. 3. Implication graph of Example 2

Suppose that the fifth clause is the conflicting clause. Conflict analysis goes back through the implication graph building the resolution tree shown in Fig. 4. The tree is comprised of leaves, c_i, which correspond to the clauses appearing in the implication graph; intermediate nodes, γ_i, which are the resolvents, and a root node which is both a resolvent and the conflict-learned clause. The resolution tree shows that γ_2 subsumes c_3, and that γ_4 subsumes c_1. The subsumed clauses can be strengthened by eliminating the pivot variable on which they were resolved. In addition to the simplifications, γ_4 containing the first UIP subsumes c_1; it is not required to add it to the clause data base. Rather, c_1 is strengthened. The simplified CNF is therefore

$$F' = (a \vee b) \wedge (a \vee b \vee \neg d) \wedge (c \vee d) \wedge (c \vee e \vee f) \wedge (d \vee e \vee \neg f) \wedge (\neg b \vee \neg d \vee e) \wedge (\neg d \vee \neg e)$$

Then, the solver backtracks to level 1, which is the highest decision level in c_1. After backtracking, $b@1$ is asserted by c_1.

This example shows how the CNF database can be simplified by checking subsumption on-the-fly. First of all, a clause can be shortened when it is resolved during conflict analysis and it is subsumed by the resolvent. The resolvent may contain a UIP. Then, the clause that is strengthened can serve as conflict-learned clause. In this case, the SAT solver has the same deductive power, even without adding conflict clause. □

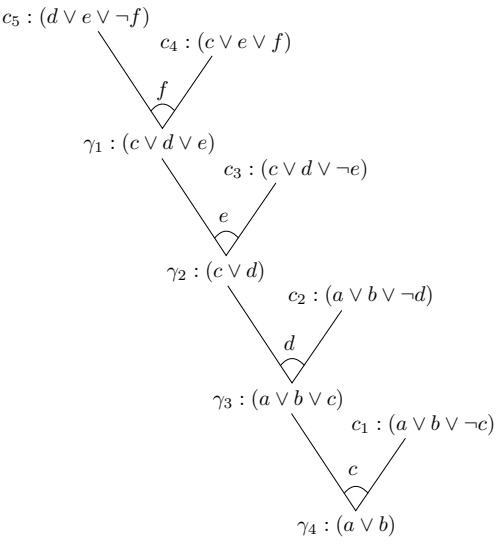


Fig. 4. Resolution tree of conflict analysis for Fig. 2

In our solver, conflict analysis based on 1-UIP may be followed by strong conflict analysis. We now consider the on-the-fly subsumption check with respect to strong conflict analysis.

Lemma 2. *Let C be a clause simplified with the resolvent produced in conflict analysis. Then C is conflicting at the current level.*

Proof. Every resolvent produced in conflict analysis is conflicting at the current decision level. Therefore, clause C, which is one such resolvent, is also conflicting. □

Lemma 3. *Let C be the clause most recently simplified by on-the-fly subsumption during conflict analysis. The subgraph of the implication graph between this clause and the 1-UIP is either a single vertex or a valid implication graph (hence, suitable for strong conflict analysis).*

Proof. The requirement for a valid implication subgraph is that the source vertex be a clause with at least two literals assigned at the current level. By Lemma 2, C is conflicting at the current decision level. If C contains the 1-UIP, the subgraph consists of a single vertex and strong conflict analysis is not invoked. Otherwise, since the residual clauses beyond C on the graph were not touched, they are also valid to be examined by strong conflict analysis. □

Lemmas 2 and 3 allow us to conclude that on-the-fly subsumption check is compatible with strong conflict analysis. As an alternative, one could postpone the strengthening of the clauses until after strong conflict analysis. Our experiments, however, indicate that it would not be as efficient.

```
1  AnalyzeConflictWithDistill(F, conflicting) {
2      resolvent = conflicting;
3      in_CNF_resolvent = TRUE;
4      while (!FoundUIP(resolvent)) {
5          lit = GetLatestAssignedLiteral(resolvent);
6          ante = GetAntecedentClause(lit);
7          var = Variable(lit);
8          resolvent′ = Resolve(resolvent, ante, var);
9          if (in_CNF_resolvent) {
10             if (Size(resolvent′) < Size(resolvent)) {
11                 StrengthenClause(resolvent, var);
12                 in_CNF_resolvent = TRUE;
13                 if (Size(resolvent′) < Size(ante))
14                     DeleteClause(ante);
15             }
16         }
17         else if (Size(resolvent′) < Size(ante)) {
18             StrengthenClause(ante, var);
19             in_CNF_resolvent = TRUE;
20         }
21         else
22             in_CNF_resolvent = FALSE;
23         resolvent = resolvent′;
24     }
25     if (!in_CNF_resolvent)
26         AddConflictClause(resolvent);
27     blevel = ComputeHighestLevelInConflictClause(resolvent);
28     return (blevel);
29 }
```

Fig. 5. Algorithm for conflict analysis with subsumption check

Returning to Example 1, in Fig. 2, γ_1, γ_2, and γ_3 all have a chance to be additional conflict clauses, since all of them have only two literals, and both literals are assigned at the current level. The strong conflict analysis, however, dismisses γ_1 because it is too close to the conflicting clause. Therefore, it misses the chance to strengthen c_6 and drop c_7. In contrast, on-the-fly subsumption is not constrained to use only one clause for simplification.

Figure 5 shows the pseudocode of the algorithm that detects and simplifies the subsumed clauses during conflict analysis. The algorithm AnalyzeConflictWithDistill() keeps checking the subsumption condition whenever a new resolvent is produced as long as FoundUIP() is false (line 4). By Lemma 1, if the operand exists in the clause database, that is, the old resolvent with in_CNF_resolvent = TRUE (line 9) or the antecedent (line 13 and 17), and the new resolvent contains fewer literals than one of its operands (line 10, 13, and 17), the operand is strengthened for the pivot variable by StrengthenClause() (line 11 and 18). When both operands are subsumed, only one of them is selected to survive, and the other is deleted (line 14). If a clause is replaced with the resolvent, the flag in_CNF_resolvent is set to TRUE (line 12, and 19).

Otherwise,in_CNF_resolvent is reset to FALSE (line 22), since the new resolvent does not exist in the clause database yet. At the end of the resolution step (line 25), if the final resolvent containing the UIP is identified as an existing clause, that is in_CNF_resolvent is false, the conflict analysis algorithm refrains from adding a new conflict clause into the clause database (line 26). Whether a new conflict clause is added or not, the DPLL procedure backtracks up to the level returned by the conflict analysis (line 28), and asserts the clause finally learned from the latest conflict.

The pseudocode of Fig. 5 omits some details for the sake of clarity. In the actual implementation, the implication graph is shrunk with a new conflicting clause by replacing the current conflicting clause with a newly strengthened clause, which must be a new resolvent. The modified graph then is available for strong conflict analysis.

4 Application to Preprocessors

Resolution is the main operation in preprocessing. In this section, we review its application to the preprocessors for variable elimination and clause distillation.

To select variables to be eliminated, all the variables are sorted by a metric such that $\delta = (|\text{clauses}_v| * |\text{clauses}_{\neg v}|) - (|\text{clauses}_v| + |\text{clauses}_{\neg v}|)$, where clauses_v stands for an occurrence list of variable v, and $|\text{clauses}|$ represents the length of the list. δ stresses the fact that the less symmetric occurrence lists are, the earlier the variable should be selected. The length of a resolvent should also be taken into account, because clauses may also be lengthened through resolution. This can be harmful to the SAT solver. Hence, we use an additional criterion, the number of literals of the resolvents, to choose variables to be eliminated.

To eliminate a variable, resolutions are applied to all the pairs of clauses in the occurrence lists of the two literals of the variable. In our variable elimination, all the literals of each clause are sorted by variable index. Taking the union of two sorted clauses can be done in linear time by a variation of the *merge-sort* algorithm. This linear operation guarantees that all the literals are still sorted after merging. With minor modification in the algorithm, the linear operation can be also used to check subsumption relation between two clauses.

A variable is eliminated only when the produced resolvents are fewer than the occurrence clauses of the variable. At each resolution operation, we can check if one of the operands is subsumed by the resolvent, like the on-the-fly subsumption check in conflict analysis of Sect. 3. A clause can be simplified by the on-the-fly subsumption, regardless of whether the variable is eliminated. The clause simplified by the on-the-fly subsumption is removed out of the occurrence list. In such a case, the current elimination check may benefit from the shortened occurrence list. Every simplified clause is checked for subsumption to other clauses after the variable elimination check.

During distillation of clauses, conflict analysis takes the majority of the time. Conflict analysis in clause distillation also performs resolutions steps as conflict analysis in DPLL. Therefore we can increase efficiency in conflict analysis by using on-the-fly simplification. In the original algorithm of distillation, the clauses of the SAT instance are distilled only if such conflict clauses are detected. More chances for simplifications, however, can be produced if we apply on-the-fly simplification. In addition to

the on-the-fly subsumption check, the distillation procedure is combined with "regular subsumption check" used in variable elimination procedure. The clause that is being distilled may participate in conflict analysis. If a clause that is an antecedent in conflict analysis contains all the decision variables, then it is the clause being distilled. We can identify such clauses while scanning the clause for resolution. If such a clause is found, it can be replaced by the conflict clause. Otherwise, the regular subsumption check can simplify the clause after distillation procedure.

5 Experimental Results

We have implemented the three applications of on-the-fly subsumption checking proposed in this paper on top of the CirCUs SAT solver [5]. The implemented approaches are the enhanced variable elimination, the improved Alembic, and on-the-fly simplification in conflict analysis.

The benchmark suite is composed of all the instances (no duplication) from the industrial category of the SAT Races of 2006 and 2008, and the SAT competition of 2007. We conducted the experiments on a 2.4 GHz Intel Core2 Duo processor with 4GB of memory. We used 3600 seconds as timeout, and 2GB as memory bound. We tested MiniSat 2.0 along with CirCUs 2.0 to provide a reference point.

Figure 6(a) shows the CPU time taken by MiniSat and CirCUs with and without their respective preprocessors. In the plot, to distinguish from SatELite, our variable elimination algorithm only is named EV, and EV with clause distillation of Alembic is named EVAL, and OTS stands for on-the-fly simplification. The data points on the plot show how many instances are solved within the given time bound.

Figure 6(b) details the effect of adding on-the-fly subsumption check to each of the three steps discussed in this paper. It can be seen that while the overall benefit is clear, there is no advantage to the application of on-the-fly subsumption to conflict analysis alone. We investigate this apparent anomaly with the help of Fig. 7, which compares the number of on-the-fly subsumptions per conflict(Fig. 7(a)) and the average resolution depth in conflict analysis(Fig. 7(b)) with and without variable preprocessing. The scatterplot confirms that there is a strong synergy between preprocessing and on-the-fly subsumption. Further analysis shows that variable elimination is the main responsible for the marked increase of subsumptions per conflict and for the shortening of resolution steps computed in conflict analysis. The following example sheds light on this phenomenon.

Example 3. Consider the following clauses:

$$(\neg a \vee \neg p)_1 \wedge (b \vee \neg p)_2 \wedge (a \vee \neg b \vee p)_3 \wedge (a \vee \neg q)_4 \wedge (\neg b \vee \neg q)_5 \wedge (\neg a \vee b \vee q)_6$$
$$\wedge(\neg p \vee r)_7 \wedge (\neg q \vee r)_8 \wedge (p \vee q \vee \neg r)_9 \wedge (a \vee \neg s)_{10} \wedge (b \vee \neg s)_{11} \wedge (\neg a \vee \neg b \vee s)_{12}$$
$$\wedge(\neg a \vee \neg t)_{13} \wedge (\neg b \vee \neg t)_{14} \wedge (a \vee b \vee t)_{15} \wedge (\neg s \vee \neg u)_{16} \wedge (\neg t \vee \neg u)_{17}$$
$$\wedge(s \vee t \vee u)_{18} \wedge (r \vee u)_{19} \wedge (\neg r \vee \neg u)_{20}.$$

Suppose that the SAT solver makes decisions $\neg a@1$ and $\neg b@2$. This leads to a conflict on c_{19}, with the implication graph shown in Fig. 8. There are no instances of

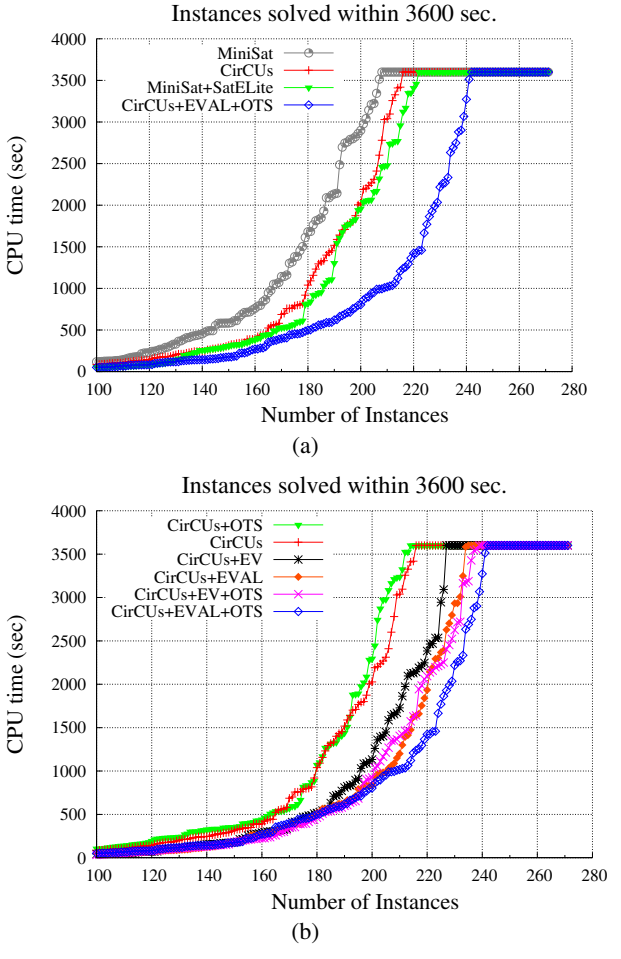


Fig. 6. Number of instances solved by various SAT solvers versus CPU time. (a) comparison of the proposed algorithm to modern SAT solvers; (b) individual contributions of simplification methods to CirCUs.

on-the-fly subsumption during conflict analysis, even though the learned clause, $\gamma_5 = a \vee b$, subsumes c_{15}: γ_5 directly subsumes other resolvents rather than c_{15}.

If we eliminate p, q, s, and t, we get the following clauses:

$$(a \vee \neg b \vee r)_1 \wedge (a \vee b \vee \neg r)_2 \wedge (\neg a \vee \neg b \vee \neg r)_3 \wedge (\neg a \vee b \vee r)_4 \wedge (a \vee \neg b \vee u)_5$$
$$\wedge (a \vee b \vee \neg u)_6 \wedge (\neg a \vee \neg b \vee \neg u)_7 \wedge (\neg a \vee b \vee u)_8 \wedge (r \vee u)_{14} \wedge (\neg r \vee \neg u)_{15} \ .$$

Figure 9 shows that the conflict-learned clause subsume c_2. (It also subsumes c_6, but this is not detected by the algorithm.) This time there are fewer resolution steps, and this "abridgment" of the process allows the subsumed clause to enter the analysis right

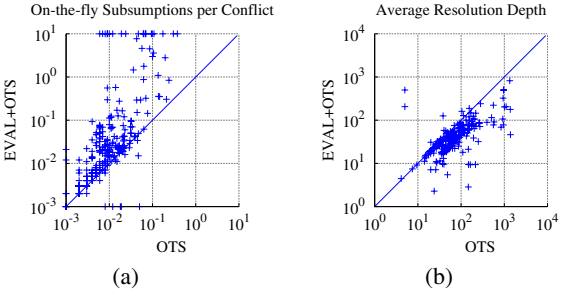


Fig. 7. (a) On-the-fly subsumptions per conflict; (b) Average depth of resolution

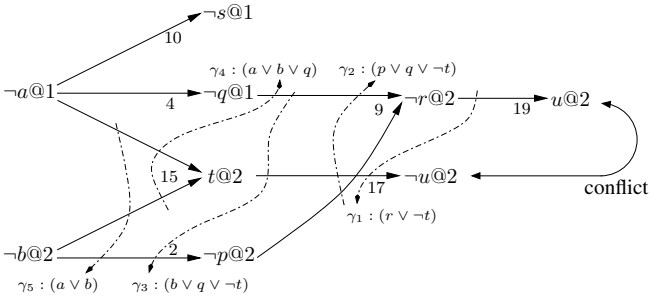


Fig. 8. Implication graph for the original clauses of Example 3

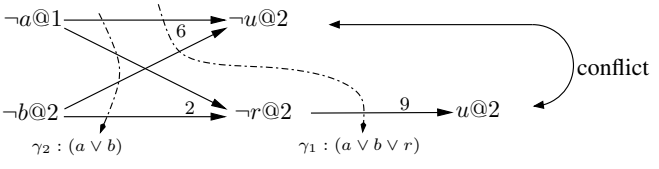


Fig. 9. Implication graph for the clauses of Example 3 after elimination

before the subsuming resolvent is computed instead of several steps before. This mechanism seems to account for several cases in which variable elimination, and preprocessing in general, increase the frequency of on-the-fly subsumptions. □

For the purpose of calibration, we also compared CirCUs+EVAL+OTS to Rsat 3.01 [8]. In the comparison, CirCUs+EVAL+OTS solved 240 instances, and Rsat 3.01 solved 256 instances within one hour.

Finally, we report statistics on the performance of the preprocessors. Figure 10(a) compares the speed of EVAL to SatELite. In this case, SatELite is run on all CNF formulae, while, in Fig. 6(a), SatELite can be disabled depending on the size of CNF formulae. This results in a few timeouts, but otherwise the two preprocessors are quite

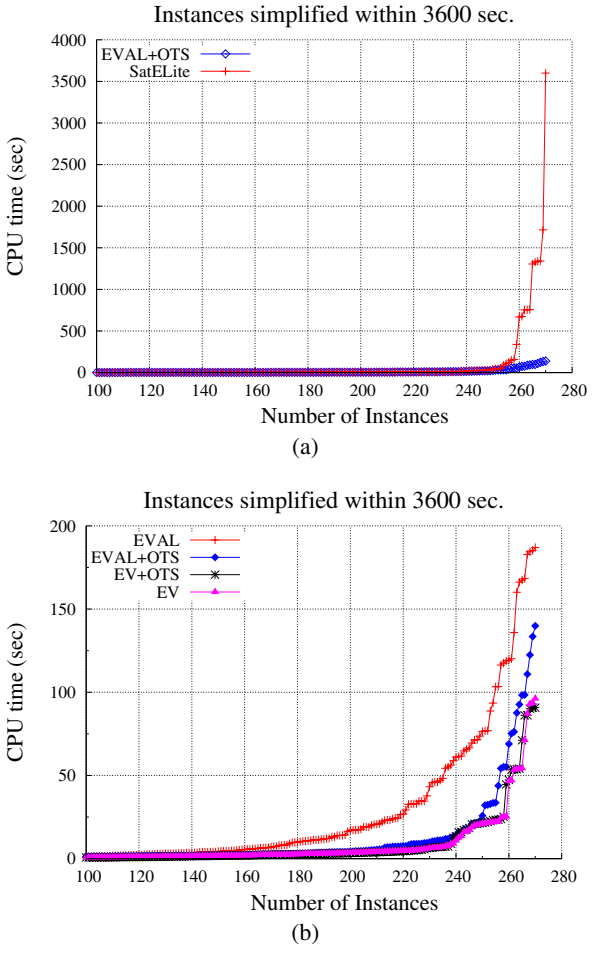


Fig. 10. Number of instances simplified by various preprocessors versus CPU time. (a) comparison of the proposed preprocessor to SatELite in MiniSat; (b) detailed comparison of the applied preprocessing techniques.

comparable. It should be noted that the current implementation of Alembic is much faster than that described in [4]. Figure 10(b), in particular, shows that on-the-fly subsumption significantly contributes to the improved preprocessor speed.

It is also interesting to compare the reductions achieved by the different preprocessors. In Fig. 11, we see that CirCUs's variable elimination is less aggressive than SatELite: it eliminates fewer clauses, but almost never increases the number of literals. Adding Alembic yields the least number of clauses without compromising the good performance in terms of literals.

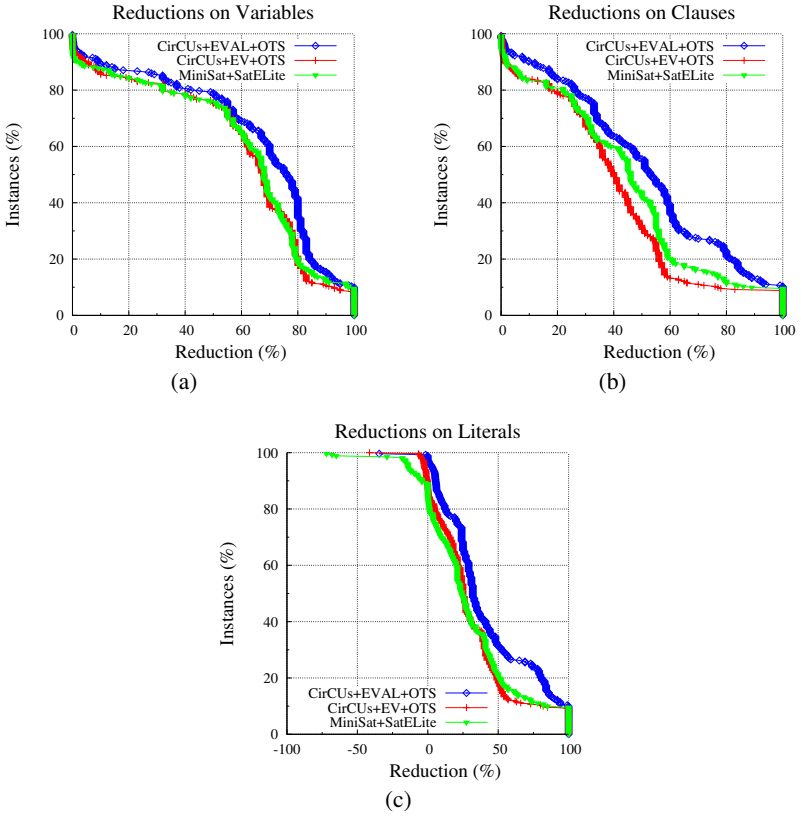


Fig. 11. Ratio of simplification made by various preprocessors on variables, clauses, and literals

6 Conclusions

We have presented a simple, efficient technique to detect subsumption of an operand by the resolvent of two clauses. This technique can be applied with minimal overhead to both preprocessing of the CNF formula and to conflict analysis. The effect is to simplify clauses in such a way that implications are obtained earlier and conflicts are sometimes avoided. Another beneficial effect is the reduction of the number of added conflict-learned clauses without detriment for the deductive power. Our experiments show that the new technique delivers a significant improvement in performance. One final advantage is its compatibility with advanced conflict analysis techniques and with the generation of unsatisfiability proofs.

References

[1] Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)

[2] Eén, N., Mishchenko, A., Sörensson, N.: Applying logic synthesis for speeding up SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 272–286. Springer, Heidelberg (2007)
[3] Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
[4] Han, H., Somenzi, F.: Alembic: An efficient algorithm for CNF preprocessing. In: Proceedings of the Design Automation Conference, San Diego, CA, June 2007, pp. 582–587 (2007)
[5] Jin, H., Awedh, M., Somenzi, F.: CirCUs: A satisfiability solver geared towards bounded model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 519–522. Springer, Heidelberg (2004)
[6] Jin, H., Somenzi, F.: Strong conflict analysis for propositional satisfiability. In: Design, Automation and Test in Europe (DATE 2006), Munich, Germany, March 2006, pp. 818–823 (2006)
[7] Nadel, A.: The Jerusat SAT solver. Master's thesis, Hebrew University of Jerusalem (2002)
[8] http://reasoning.cs.ucla.edu/rsat/
[9] http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat.html
[10] Silva, J.P.M., Sakallah, K.A.: Grasp—a new search algorithm for satisfiability. In: Proceedings of the International Conference on Computer-Aided Design, San Jose, CA, November 1996, pp. 220–227 (1996)
[11] Sörensson, N., Eén, N.: MiniSat v1.13 – a SAT solver with conflict-clause minimization. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569. Springer, Heidelberg (2005)
[12] Zhang, L.: On subsumption removal and on-the-fly CNF simplification. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 482–489. Springer, Heidelberg (2005)
[13] Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in Boolean satisfiability solver. In: Proceedings of the International Conference on Computer-Aided Design, San Jose, CA, November 2001, pp. 279–285 (2001)

Dynamic Symmetry Breaking by Simulating Zykov Contraction

Bas Schaafsma, Marijn J.H. Heule*, and Hans van Maaren

Department of Software Technology, Delft University of Technology
schaafsma@ch.tudelft.nl, marijn@heule.nl, h.vanmaaren@tudelft.nl

Abstract. We present a new method to break symmetry in graph coloring problems. While most alternative techniques add symmetry breaking predicates in a pre-processing step, we developed a learning scheme that translates each encountered conflict into *one* conflict clause which covers equivalent conflicts arising from *any* permutation of the colors.

Our technique introduces new Boolean variables during the search. For many problems the size of the resolution refutation can be significantly reduced by this technique. Although this is shown for various hand-made refutations, it is rarely used in practice, because it is hard to determine which variables to introduce defining useful predicates. In case of graph coloring, the reason for each conflicting coloring can be expressed as a node in the Zykov-tree, that stems from merging some vertices and adding some edges. So, we focus on variables that represent the Boolean expression that two vertices can be merged (if set to true), or that an edge can be placed between them (if set to false). Further, our algorithm reduces the number of introduced variables by reusing them.

We implemented our technique in the state-of-the-art solver minisat. It is competitive with alternative SAT based techniques for graph coloring problems. Moreover, our technique can be used on top of other symmetry breaking techniques. In fact, combined with adding symmetry breaking predicates, huge performance gains are realized.

1 Introduction

Satisfiability (SAT) solvers have become very powerful in recent years. Especially conflict-driven clause learning SAT solvers can effective tackle certain huge problems. Crucial to strong performance is learning *conflict clauses* that ensure that the same search space is not explored multiple times. However, in the presence of symmetry the effectiveness of conflict clauses is highly reduced: search spaces could be visited that are symmetric to already refuted areas.

This paper focusses on symmetry in graph coloring problems. In particulary, we want to break the symmetry that arises by permuting the colors. This can be broken *statically*, as a preprocessing step, or *dynamically*, during the search. A frequently used static technique assigns a different color to all vertices in a large

* Supported by the Dutch Organization for Scientific Research (NWO) under grant 617.023.611.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 223–236, 2009.

clique [19]. Although effective and cheap (a large clique is easy to find), it only breaks the symmetry partially [15]. A dynamic symmetry breaking technique [10] adds, besides the conflict clause expressing the conflict, all symmetric conflict clauses. Yet the number of symmetric conflict clauses grows exponentially with the number of colors. Here, we present an alternative dynamic technique.

For each conflicting assignment of k colors in the DPLL-tree there exists $k!$ symmetric conflicting assignments that can be obtained by a permutation of the colors. At the core of our algorithm is the observation that all these symmetric conflicting assignments correspond to the same node in the Zykov-tree: a binary search tree that selects in each node two nonadjacent vertices of the graph being colored. One branch explores the search space by merging these vertices (the same color), while the other branch examines the space created by placing an edge between them (not the same color). We transform each conflicting DPLL-node to the corresponding Zykov-node and translate the latter back to SAT.

Since the Zykov algorithm branches on merging two nonadjacent vertices or placing an edge between, new variables are introduced – called *merge variables*: these variables represent that two vertices must have the same color (a merge step) if set to true, while they must be colored differently (adding an edge) if set to false. The proposed technique converts the original variables in conflict clauses to merge variables.

The outline of this paper is as follows: Section 2 deals with encoding graph coloring problems into SAT. Transformation of conflict clauses is explained in Section 3. Section 4 offers experimental results. Finally, in Section 5 we draw some conclusions and provide suggestions for future research.

2 Preliminaries

2.1 The Satisfiability Problem

The Satisfiability problem (in short SAT) asks whether there exists an assignment for a given Boolean formula such that it evaluates to *true*. If such an assignment does exist, we call the problem satisfiable else the problem is qualified as unsatisfiable. In a more formal setting a formula $\mathcal{F} = \{C_1 \wedge \ldots \wedge C_m\}$ consists of conjunction of clauses C_i, while each clause $C_i = (l_{i,1} \vee \ldots \vee l_{i,j})$ consists of disjunction of literals. A literal l refers either to a Boolean variable x_i or to its negation $\neg x_i$. A clause is satisfied when at least one of its literals evaluates to *true*. Finally, a satisfying assignment satisfies all clauses.

2.2 The k-Coloring Problem

The k-coloring problem deals with the question whether the vertices of a graph can be colored with k colors such that two connected vertices have a different color. Or more formal, let φ_{color} be a mapping of vertices $v \in V$ onto an integer in $\{1, \ldots, k\}$. A graph $G = (V, E)$ is k-colorable, when there exists a φ_{color} to *all* vertices such that for every $(v, w) \in E$, $\varphi_{\text{color}}(v) \neq \varphi_{\text{color}}(w)$. The smallest k for which G is still k-colorable is known as the chromatic number of G or $\mathcal{X}(G)$.

The k-coloring problem can be naturally translated to SAT. We focus on the widely used *direct encoding* [14]. It uses Boolean variables $x_{v,i} \leftrightarrow \varphi_{\text{color}}(v) = i$, which we refer to as *color variables*. The property that all vertices must be colored is encoded by the *at-least-one* clauses, which are of the form:

$$\bigwedge_{v \in V} (x_{v,1} \vee x_{v,2} \vee \cdots \vee x_{v,k}) \tag{1}$$

Further, for each $(v, w) \in E$, k *conflicting clauses* encode $\varphi_{\text{color}}(v) \neq \varphi_{\text{color}}(w)$:

$$\bigwedge_{1 \leq i \leq k} \bigwedge_{(v,w) \in E} (\neg x_{v,i} \vee \neg x_{w,i}) \tag{2}$$

The above is known as the *minimum encoding* [9]. The *extended encoding* adds redundant clauses which encode that vertices must have *at-most-one* color:

$$\bigwedge_{1 \leq i < j \leq k} \bigwedge_{v \in V} (\neg x_{v,i} \vee \neg x_{v,j}) \tag{3}$$

Although optional, most complete solvers perform better on instances where these clauses have been added [14]. Yet for our technique they are not required.

2.3 Zykov Contraction Algorithms

One of the main family of algorithms which determines $\mathcal{X}(G)$ for a graph G, or approximates $\mathcal{X}(G)$ is known as Contraction. This family of algorithms is based on a theorem due to Zykov [20], which states:

$$\mathcal{X}(G) = min(\mathcal{X}(G/(v, w)), \mathcal{X}(G + (v, w))) \tag{4}$$

In this theorem, $G/(v, w)$ denotes the graph with vertex v and w contracted, meaning that vertex w is deleted and all its edges are transferred to v. $G+(v, w)$ means that an edge is added between vertex v and w, as shown in Figure 1.

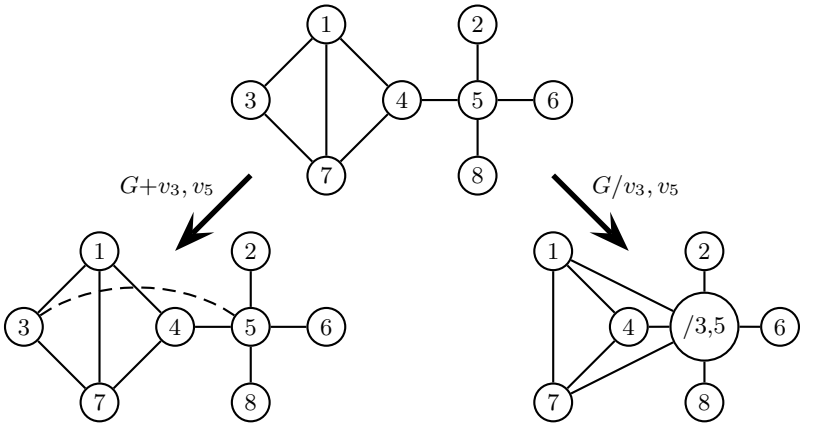


Fig. 1. A Zykov-tree example. The numbers in the vertices refer to their index v_i.

Repeated steps of applying this theorem to a graph G result in a binary tree. The leaves of this tree are fully connected graphs, which each have a chromatic number equal to their number of vertices. The chromatic number of G is then equal to the chromatic number of the graph with the least amount of vertices.

Zykov Contraction can be simulated in a SAT solver by adding redundant variables and clauses to the CNF translation of a graph coloring problem. Adding redundant variables and clauses was introduced by Tseitin and is known as *Extended Resolution* (ER) [17]. ER is shown to be very powerful in theory [3].

Each step of the Contraction algorithm can be simulated by introducing a Boolean variable $e_{v,w}$, referred to as *merge variables*, which expresses:

$$e_{v,w} \leftrightarrow \varphi_{\text{color}}(v) = \varphi_{\text{color}}(w) \tag{5}$$

This relation can be translated to CNF using the following clauses:

$$\bigwedge_{1 \leq i \leq k} (e_{v,w} \vee \neg x_{v,i} \vee \neg x_{w,i}) \wedge (\neg e_{v,w} \vee x_{v,i} \vee \neg x_{w,i}) \wedge (\neg e_{v,w} \vee \neg x_{v,i} \vee x_{w,i}) \tag{6}$$

These clauses will propagate the fact that vertices v and w have equal or unequal colors when $e_{v,w}$ is set. If set to true the clauses simulate merging two vertices, while setting $e_{v,w}$ to false represents placing an edge between them.

Initially, we studied the use of adding merge variables and the corresponding clauses to a given formula as a preprocessing step. This turned out to merely decrease the performance. However, we observed that one could capitalize on the expressive power of merge variables by strengthening conflict clauses. Therefore, instead of ER, only the clauses are added which are required for the Tseitin translation [17] of these learnt clauses.

3 Merge Clauses

A powerful application of simulating Contraction lies in strengthening conflict clauses in conflict-driven algorithms [11] for graph coloring instances. Simply put, conflict-driven solvers continue to assign variables until a conflict is detected. When a conflict is detected, the solver determines an assignment responsible for this conflict and adds a conflict clause C_{conflict} to $\mathcal{F}$, where C_{conflict} is the negation of the assignments which led to the conflict.

To illustrate the benefits of simulating Contraction, consider the example conflict, in the 3-coloring instance presented in Figure 2. In the corresponding SAT instance, a conflict clause for this conflict would be:

$$(\neg x_{1,1} \vee \neg x_{2,1} \vee \neg x_{3,2} \vee \neg x_{4,2} \vee \neg x_{5,3}) \tag{7}$$

Yet, due to the inherent symmetries of a k-coloring instance, any permutation of colors in a conflicting assignment is also a conflicting assignment. Thus for the corresponding SAT instance, the following clause is also logically implied by $\mathcal{F}$:

$$(\neg x_{1,3} \vee \neg x_{2,3} \vee \neg x_{3,1} \vee \neg x_{4,1} \vee \neg x_{5,2}) \tag{8}$$

Unfortunately, with a maximum of $k!$ possible permutations it is impractical to add each implied clause, for almost any k larger than four [10].

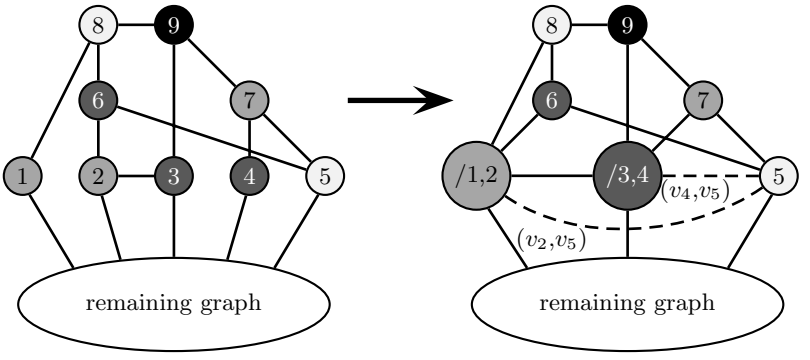


Fig. 2. A Zykov Contraction example. The numbers in the vertices refer to the index v_i. The added edges are shown as dashed lines. Vertex v_9 is in conflict because it cannot be colored. The example focusses on the assignment to v_1, v_2, v_3, v_4, and v_5.

3.1 Transforming Conflict Clauses

Any conflict clause, which consists of negative color literals, such as the clauses depicted in (7) and (8), encodes a conflicting coloring φ_{color} of a subset of vertices in G. This encoded coloring corresponds to some node in a Zykov-tree with G as root. Vertices in this subset that are equally colored in φ_{color} are contracted into a single vertex and edges are added to induce a clique among these contracted vertices. This relation exists, because once the vertices are contracted and the clique is induced, any two vertices equally colored in φ_{color}, will be equally colored in any coloring of our created clique, because they have been merged. Furthermore, any two vertices that were not equally colored in φ_{color}, will be unequally colored in any coloring of our clique, because there exists an edge between them. Thus any coloring of the created clique corresponds to a permutation of φ_{color} and therefore will, just like φ_{color}, result in a conflict. Therefore, any conflict clause consisting out of negative color literals can be converted in a *corresponding merge clause*, denoted by C_{merge}, which is a conflict clause consisting out of merge variables.

Back to the example, consider the conflict depicted in Figure 2 as a node in a Zykov-tree, in which v_1 and v_2 are merged, v_3 and v_4 are merged, and the edges (v_2, v_5), (v_4, v_5) are added. This is represented using merge variables as:

$$(\neg e_{1,2} \vee \neg e_{3,4} \vee e_{2,5} \vee e_{4,5}) \tag{9}$$

In any merge clause C_{merge}, negative literals correspond to contractions of the equally colored vertices in φ_{color}. For each set of n equally colored vertices in φ_{color} we will need $n-1$ negative merge literals. The positive literals C_{merge} correspond to the edges added to induce a clique. Of course no edges need to be added between contracted vertices v and w, if such an edge already exists in G.

Unfortunately, in most cases one could choose from many merge variables to construct a merge conflict clause. In the example, instead of using $e_{2,5}$ (or $e_{4,5}$), one could select $e_{1,5}$ (or $e_{3,5}$). The choice of the merge variables influences

the performance, therefore one would prefer to select the "optimal" candidates. Heuristics for this selection are discussed in Section 3.2.

Besides C_{merge}, one also needs to add the clauses $M(C_{\text{merge}})$, which arise from the Tseitin translation [17], to $\mathcal{F}$. Theoretically, for each introduced merge variable we could add the full set of clauses described in Section 2.3. Yet, in practice it suffices to add only the clauses that contain the negation of the literal of our introduced variable. Only adding these clauses is good practice as it saves resources [13]. For any C_{merge}, $M(C_{\text{merge}})$ equals to:

$$\bigwedge_{1\leq i\leq k}\Big(\bigwedge_{e_{v,w}\in C_{\text{merge}}}((\neg e_{v,w}\vee x_{v,i}\vee\neg x_{w,i})\wedge(\neg e_{v,w}\vee\neg x_{v,i}\vee x_{w,i}))\wedge\bigwedge_{\neg e_{v,w}\in C_{\text{merge}}}(e_{v,w}\vee\neg x_{v,i}\vee\neg x_{w,i})\Big) \tag{10}$$

Thus $M(C_{\text{merge}})$ for our example is:

$$\bigwedge_{1\leq i\leq k}\Big((e_{1,2}\vee\neg x_{1,i}\vee\neg x_{2,i})\wedge(\neg e_{2,5}\vee x_{2,i}\vee\neg x_{5,i})\wedge(\neg e_{2,5}\vee\neg x_{2,i}\vee x_{5,i})\wedge(e_{3,4}\vee\neg x_{3,i}\vee\neg x_{4,i})\wedge(\neg e_{4,5}\vee x_{4,i}\vee\neg x_{5,i})\wedge(\neg e_{4,5}\vee\neg x_{4,i}\vee x_{5,i})\Big) \tag{11}$$

3.2 Implementation

We have applied the principal of merge conflict clauses in the conflict-driven clause learning (CDCL) SAT-solver architecture which we refer to as the CDCLmerge algorithm. CDCLmerge is specialized for the k-coloring problem and uses merge conflict clauses to store conflicts. Its most important feature is the transformConflict procedure, which transforms the color literals in a conflict clause to merge literals. In order to make the transformConflict function properly, we also had to adapt the decide procedure. Algorithm 1 gives a detailed overview of the CDCLmerge algorithm.

The decide procedure
The proposed transformation to merge clauses requires that all conflicts can be expressed as a disjunctions of negative color literals and merge literals. This cannot be guaranteed if the solver branches on negative literals. E.g. consider the perfect graph of size three. Assigning $x_{1,1}$ to false, $x_{2,1}$ to false, and $x_{3,2}$ to true results in a conflict which can be expressed as $(x_{1,1}\vee x_{2,1}\vee\neg x_{3,2})$. Notice that this conflict clause cannot be translated to a merge clause is a meaningful way. Therefore, decide is adapted such that it assigns each decision variable to true. This heuristic is similar to the one used in minisat which assigns all decision variables to false [7].

The transformingConflict procedure
The input C_{conflict} is transformed into C_{merge} using the following steps:

1. Positive color literals in C_{conflict} are replaced by merge and negative color literals by expanding them into their reason literals. For instance, say the example conflict clause would have been $(x_{1,3}\vee x_{2,2}\vee x_{3,3}\vee x_{4,2}\vee\neg x_{5,3})$.

Algorithm 1. CDCLMERGE($\mathcal{F}$)

```
1: while true do
2:   PROPAGATE() /* propagate unit clauses */
3:   if not conflict then
4:     if all variables assigned then
5:       return satisfiable
6:     end if
7:     DECIDE() /*select decision variable. ADAPTED*/
8:   else
9:     C_conflict ← ANALYZE() /*analyze the conflict*/
10:    C_merge ← TRANSFORMCONFLICT(C_conflict) /*ADDED*/
11:    if top level conflict found then
12:      return unsatisfiable
13:    end if
14:    BACKTRACK(C_conflict) /*backtrack while C_conflict remains unit or falsified*/
15:  end if
16: end while
```

Assume that the same conflicting coloring was its reason. In that case $\neg x_{1,2}$ will be the reason literal for $x_{1,3}$. Therefore, we can replace the latter by the former. This process is iterated while C_{conflict} contains positive literals.

2. Redundant literals (see Theorem 2 and 3) are removed from C_{conflict}.
3. C_{conflict} is split into C_{color}, which consist out of all negative color literals in C_{conflict} and C_{extra}, which consists of all merge literals in C_{conflict}.
4. Transform C_{color} into a merge clause C_{zykov} by computing the corresponding node in the Zykov-tree. Preliminary tests showed that the performance improved if the number of introduced variables were kept to a minimum and introduced variables were reused whenever possible. Therefore, in case of choice between possible merge literals to use in the transformation to C_{zykov}, the merge literal is selected which is most frequently used in conflict clauses. Ties are broken pseudo randomly.
5. Return the union of C_{zykov} and C_{extra} as the transformed clause C_{merge}.

The BACKTRACK procedure

Conflict-driven clause learning SAT solvers backtrack (also known as backjump) to the lowest decision level where the latest conflict clause is still a unit clause. In CDCLMERGE this aspect of the solving algorithm is not changed. However, if a conflict clause C_{conflict} is unit, a corresponding merge clause C_{merge} may not be unit.

Recall the example at the start of this section:

$$C_{\text{conflict}} \Leftrightarrow C_{\text{merge}}$$
$$(\neg x_{1,1} \vee \neg x_{2,1} \vee \neg x_{3,2} \vee \neg x_{4,2} \vee \neg x_{5,3}) \Leftrightarrow (\neg e_{1,2} \vee \neg e_{3,4} \vee e_{2,5} \vee e_{4,5})$$

Say that variable $x_{v,i}$ is assigned at level v. The BACKTRACK procedure will jump to level 4. At this level C_{conflict} is reduced to $(\neg x_{5,3})$, while C_{merge} is

reduced to $(e_{2,5} \vee e_{4,5})$. The reason is that two merge literals refer to vertex v_5. Currently, this problem is solved by changing the DECIDE procedure in such a way that if the latest merge clause consists of multiple unassigned literals one of these literals is assigned to false. This is repeated until the merge clause becomes unit.

Although C_{conflict} is satisfiability equivalent to $C_{\text{merge}} \wedge M(C_{\text{merge}})$ (see Theorem 4), the transformation is not *arc-consistent* under unit propagation [8]. As soon as a merge clause contains multiple literals that refer to the same vertex, the merge clause will not become unit when the original conflict clause would be unit. In the example a similar problem would arise in case v_2 (or v_4) was the last assigned vertex, because both $\neg e_{1,2}$ and $e_{2,5}$ (or both $\neg e_{3,4}$ and $e_{4,5}$) occur in C_{merge}.

The lack of arc-consistency is a serious weakness of the current implementation. We study various options to deal with this weakness. An interesting partial solution is adding a second merge clause. Back to the example: besides C_{merge}, we could also add $(\neg e_{1,2} \vee \neg e_{3,4} \vee e_{1,5} \vee e_{3,5})$. This solves arc-consistency for vertex v_2 and v_4. However, the problem is still unsolved for vertex v_5. In general, a second merge clause can fix arc-consistency for all vertices that are colored the same as another vertex in the conflict clause.

3.3 Optimizations

Variable Selection Heuristics

Although merge variables are useful to create merge conflict clauses, they seem rather weak as decision variables. For instance, if a clique of size $k+1$ arises by assigning some merge variables (i.e. a conflicting assignment), one may not detect this at the CNF level (no empty clause). Therefore, we only branch on color (original) variables. This choice is also supported by some experiments.

Finally, we propose a specialized version of the VSIDS activity heuristic [12]. Since merge variables will not be selected as decision variables, it does not make sense to maintain an activity for them. If a merge variable should have been increased, we want to bump the activity of the corresponding color variables instead. This idea have been implemented using an activity counter for vertices too. Each time a merge variable contributes to a conflict, the activity heuristic of both corresponding vertices is increased. The selection of decision variables is narrowed by choosing a variable from the most active vertex. This variant of VSIDS is inspired by [2].

Symmetry Breaking in the Presence of Unit Clauses

In the presence of symmetry, it is good practice to add *symmetry breaking predicates* [15]. In case of graph coloring problems, one can search for a large clique and force all vertices in that clique to a different color – by adding unit clauses to the formula. Cliques in a graph can be cheaply detected using the algorithm by M. Trick [21]. In the more general context of CNF formulae, shatter [1] can be used to compute symmetry breaking predicates.

Apart from symmetry breaking predicates, many structured graph coloring problems, such as quasi-group instances [9], contain unit clauses. In case the

symmetry is already partially broken by some unit clauses, it does not make sense to introduce merge variables.

Regarding the implementation: if unit clause $(x_{u,i}) \in \mathcal{F}$ and $\neg x_{u,i} \in C_{\text{conflict}}$, then none of the literals $\neg x_{v,i} \in C_{\text{conflict}}$ are replaced by merge literals. Further, if unit clause $(x_{u,i}) \in \mathcal{F}$, then for all positive merge literals $e_{u,w}$ that would have added, the positive color literal $x_{w,i}$ is added instead.

3.4 Proof of Correctness of Merge Conflict Clauses

Definition 1. *Let $\pi : (1, \ldots, k) \to (1, \ldots, k)$ be a function that is one to one and onto.*

Definition 2. *Let $\mathcal{P}_\pi$ be a function for which holds (with $l_{h,i}$ as literals of C_h):*

$$\begin{aligned}
\mathcal{P}_\pi(C_h) &= (\mathcal{P}_\pi(l_{h,1}) \vee \ldots \vee \mathcal{P}_\pi(l_{h,i})) \\
\mathcal{P}_\pi(\neg l_{h,i}) &= \neg \mathcal{P}_\pi(l_{h,i}) \\
\mathcal{P}_\pi(x_{v,i}) &= x_{v,\pi(i)} \\
\mathcal{P}_\pi(e_{v,w}) &= e_{v,w}
\end{aligned}$$

Theorem 1. *If Boolean function $\mathcal{F}$ represents a k-coloring problem and clause C_h is logically implied by $\mathcal{F}$, then for any π, $\mathcal{P}_\pi(C_h)$ is logically implied by $\mathcal{F}$.*

Proof. Every satisfying assignment makes C_h true. Applying π to the satisfying assignments yields a permutation of them. So, these assignments satisfy $\mathcal{P}_\pi(C_h)$.

Theorem 2. *Let C_{conflict} be a conflict clause consisting of merge literals and negative color literals. Let $\mathcal{C} = \{i : \neg x_{v,i} \in C_{\text{conflict}}\}$ denote the set of colors used in C_{conflict}. A literal $\neg x_{u,i} \in C_{\text{conflict}}$ is redundant, if C_{conflict} does not contain a literal $\neg x_{v,i}$ $(u \neq v)$ and for each $j \in \mathcal{C} (j \neq i)$ C_{conflict} contains a literal $\neg x_{w,j}$ $(u \neq w)$ such that $(u, w) \in E$ (the edge set).*

Proof. C_{conflict} with and without $\neg x_{u,i}$ correspond to the same node in the Zykov-tree, because $\neg x_{u,i}$ is the only literal assigned to i assures that no merge steps are required, while no edges have to be added, because for each $j \in \mathcal{C} (j \neq i)$, u is already connected to a vertex w with $\varphi_{\text{color}}(w) = j$.

Theorem 3. *Let C_{conflict} be a conflict clause consisting of merge literals and negative color literals. A literal $\neg e_{u,v} \in C_{\text{conflict}}$ is redundant, if $(\neg x_{u,i} \vee \neg x_{v,i}) \in C_{\text{conflict}}$, while a literal $e_{u,w} \in C_{\text{conflict}}$ is redundant, if $(\neg x_{u,i} \vee \neg x_{w,j}) \in C_{\text{conflict}}$.*

Proof. Any solution to a graph coloring problem assigns a Boolean value to *all* color variables. So, each solution will be a *full assignment*. Each full assignment which satisfies $\neg e_{u,v}$ also satisfies $(\neg x_{u,i} \vee \neg x_{v,i})$, while each full assignment which satisfies $e_{u,w}$ also falsifies $(\neg x_{u,i} \vee \neg x_{w,j})$.

Notice that based on this theorem, we can conclude that a formula is unsatisfiable if a conflict clause only consists of a negative color literal. We refer to a *reduced clause* if all redundant literals (based on Theorem 2 and 3) are removed.

Theorem 4. *Let Boolean function $\mathcal{F}$ represent a k-coloring problem and let C_h be a reduced clause logically implied by $\mathcal{F}$. If C_h consists of merge literals and negative color literals and C_{merge} is a corresponging merge clause of C_h, then*

$$\bigwedge_{\pi_1..\pi_{k!}} \mathcal{P}_{\pi_i}(C_{\text{h}}) \text{ is satisfiability equivalent to } C_{\text{merge}} \wedge M(C_{\text{merge}}) \tag{12}$$

Proof. Recall that any solution must be a full assignment. (UNSAT $\Rightarrow$ UNSAT) If a full assignment falsifies $\bigwedge \mathcal{P}_{\pi_i}(C_h)$, then there exists a π_i for which $\mathcal{P}_{\pi_i}(C_h)$ is falsified. Since $\mathcal{P}_{\pi_i}(C_{\text{merge}}) = C_{\text{merge}}$ also represents $\mathcal{P}_{\pi_i}(C_h)$, $C_{\text{merge}} \wedge M(C_{\text{merge}})$ is falsified as well. (SAT $\Rightarrow$ SAT) If a full assignment satisfies $\bigwedge \mathcal{P}_{\pi_i}(C_h)$ by merge literals in C_h, then C_{merge} is also satisfied because it contains all merge literals in C_h. Notice that because C_h is a reduced clause, it either contains zero negative color literals (in case the former case is applicable) or at least two negative color literals. A full assignment can only satisfy $\bigwedge \mathcal{P}_{\pi_i}(C_h)$ if two vertices are assigned a different color while the corresponding color literals in C_h have the same color index, or two vertices are assigned the same color while the corresponding color literals in C_h have the different color index. In both cases $C_{\text{merge}} \wedge M(C_{\text{merge}})$ is satisfied as well.

Thus once we have learnt C_{conflict}, we could add all clauses $\mathcal{P}_{\pi_i}(C_{\text{conflict}})$ to $\mathcal{F}$ (Theorem 1). Yet, based on Theorem 4, we add $C_{\text{merge}} \wedge M(C_{\text{merge}})$ instead. Furthermore, Theorem 4 implies that using merge conflict clauses requires that every conflict can be expressed into merge literals and negative color literals. In order to ensure this, the variables selection heuristics of the solver have to be adapted. This adaptation is described in Section 3.2.

4 Results

All experiments were performed on a 2.0 GHz Intel Core 2 Duo with 1 GB of DDR2 Memory. Instances were encoded using the extended direct encoding and we used the method of finding and forcing cliques as symmetry breaking method.

4.1 Medium Sized Random Graphs

This experiment was performed to compare our CDCLMERGE implementation, referred to as MiniColor to the standard distribution of MiniSat2, branching on positive variables. In this experiment we generated 45 random graphs of 70 vertices, with varying edge probabilities (denoted by P_{edge}). Per graph, one SAT instance (G, $\mathcal{X}(G)$) and one UNSAT instance (G, $\mathcal{X}(G)$ -1) were created. Table 1 shows the average solving times and the number of solved instances.

As can be seen in Table 1 the performances on satisfiable instances are on par, although MiniColor was able to solve one more instance. On the other hand, performance on unsatisfiable instances has significantly improved. Besides solving more instances, MiniColor was on average one order of magnitude faster.

Table 1. Average runtimes for medium sized graphs, with a 1200 (s) timeout

				SAT instances				UNSAT instances			
				Minisat2		MiniColor		Minisat2		MiniColor	
$\|G\|$	$\|V\|$	P_{edge}	$\mathcal{X}(G)$	time (s)	#	time (s)	#	time (s)	#	time (s)	#
15	70	0.5	11-12	25.59	15	13.94	15	190.72	14	39.98	15
15	70	0.7	17-18	24.73	13	43.41	14	307.88	8	26.1	14
15	70	0.9	27-28	0.73	15	0.16	15	19.00	13	0.95	15

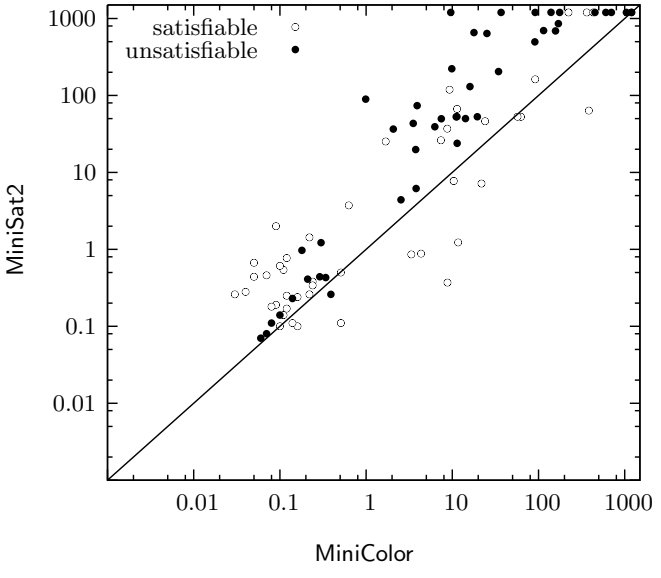


Fig. 3. Performance comparison between MiniColor and MiniSat2 on medium sized random graphs, with a 1200 (s) timeout

4.2 DIMACS Benchmarks

This experiment was executed to compare MiniColor to published results on graph coloring and to the unmodified MiniSat2 solver. As published benchmark performances we used the results published in [19] by Van Gelder which, to our knowledge, present the best broad overview of SAT based graph coloring results. Of the 27 graphs used in this benchmark set most are relatively easy. However, the five graphs presented in Table 2 were shown to be particularly difficult.

A comparison of MiniColor with best presented runtimes in [19], denoted by VanGelder and the runtimes of MiniSat2 on these graphs can be found in Table 3 and 4. For comparative purposes, we scaled the times presented in [19] to what they would have been if the instances were run on our platform[1].

[1] The `dfmax` benchmark takes 12s for `r500.5.b` on our platform compared to 16.96 in [19]. For more information of the `dfmax` benchmark, please check [22].

Table 2. Difficult DIMACS instances

instance	$\|V\|$	$\|E\|$	$\mathcal{X}$	found clique size
`Myciel6`	95	755	7	2
`Myciel7`	191	2360	8	2
`abb313GPIA`	1557	53356	9	6
`DSJC125.5`	125	3891	?	10
`DSJC125.9`	125	6961	?	33

Table 3. Runtimes on difficult satisfiable DIMACS in seconds

	SAT instances			
instance	k	VanGelder	MiniSat2	MiniColor
`Myciel6`	7	0	0	0.01
`abb313GPIA`	9	1256	3.63	1.89
`DSJC125.5`	19	4446	43.46	18.51
`DSJC125.9`	46	19119	140	16.73

Table 4. Runtimes on difficult unsatisfiable DIMACS in seconds

	UNSAT instances			
instance	k	VanGelder	MiniSat2	MiniColor
`Myciel6`	6	2113	3096	1726
`abb313GPIA`	8	5.63	0.73	0.72
`DSJC125.5`	12	488	5.85	4.08
`DSJC125.9`	37	4630	934	53.06

Table 5. Runtimes of MiniColor and MiniSat2 on harder versions of the DIMACS instances

	SAT instances			UNSAT instances		
instance	k	MiniSat2	MiniColor	k	MiniSat2	MiniColor
`Myciel7`	8	0	0.03	6	6497	1381
`DSJC125.5`	18	$>$ 19000	$>$ 19000	13	$>$ 19000	2931
`DSJC125.9`	45	$>$ 19000	1008	38	$>$ 19000	4683

As can be seen in Table 3 and 4, the runtimes of our implementation are vast improvements over the runtimes of MiniSat2 and those presented in [19]. After these encouraging results, we tried how our implementation would handle more difficult coloring of these graphs. As it turned out we could prove that `Myciel7` is not 6 colorable, `DSJC125.5` is not 13 colorable and `DSJC125.9` is not 38 colorable within reasonable time. The corresponding runtimes are shown in Table 5.

5 Conclusions and Future Research

We showed how a SAT conflict-driven solver can be optimized for graph coloring problems by converting conflict clauses in such a way that they cover all permutations of the colors. This technique can be used in combination with other optimizations for graph coloring such as adding symmetry breaking predicates. In fact, the best performances are achieved by this combination.

We introduce new Boolean variables during the search. Although very powerful in theory, it is hardly used in practice. Regarding its practical application, we learnt two lessons. First, the introduced variables should be meaningful within the context of the problem – in this case, the branches in the Zykov-tree. Second, reusage of introduced (merge) variables is crucial. Recall that in each conversion step one can choose from many merge variables. Yet, heuristics that try to minimize the number of introduced variables were required to make the technique competitive.

Although the proposed technique is, as presented, only applicable to graph coloring problems, we have reason to believe that it can be generalized. Many multi-valued SAT problems seem fit for this purpose. In particular those consisting of constraints in which variables should either have the same value or a different one. Examples of such applications are computing Van der Waerden numbers and Schur numbers. The usefulness of our ideas will depend on whether they can be generalized successfully.

Acknowledgements

The authors thank the anonymous reviewers for their valuable comments that helped improving this paper.

References

1. Aloul, F.A., Sakallah, K.A., Markov, I.L.: Efficient Symmetry Breaking for Boolean Satisfiability. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 271–282 (2003)
2. Ansótegui, C., Larrubia, J., Li, C.M., Manyà, F.: Exploiting multivalued knowledge in variable selection heuristics for SAT solvers. Annals of Mathematics and Artificial Intelligence 49(1-4), 191–205 (2007)
3. Cook, S.A.: A short proof of the pigeonhole principle using extended resolution. SIGACT News 8(4), 28–32 (1976)
4. Cook, S.A.: Feasibly constructive proofs and the propositional calculus. In: Proceedings of STOC 1975, pp. 83–97 (1975)
5. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM 5(7), 394–397 (1962)
6. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3), 201–215 (1960)
7. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)

8. Gent, I.P.: Arc Consistency in SAT. In: Proceedings of the Fifteenth European Conference on Artificial Intelligence, ECAI 2002 (2002)
9. Gomes, C.P., Shmoys, D.B.: Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In: Proceedings of the Computational Symposium on Graph Coloring and Generalizations, Ithaca, USA, pp. 22–39 (2002)
10. Keur, A., Stevens, C., Voortman, M.: Symmetry Breaking Options in Conflict Driven SAT Solving. TU-delft technical report, http://www.st.ewi.tudelft.nl/sat/reports.php
11. Marques-Silva, J.P., Sakallah, K.A.: GRASP – a new search algorithm for satisfiability. In: International Conference on Computer-Aided Design, pp. 220–227 (1996)
12. Moskewicz, M.W., Madigan, C.F.: Chaff: engineering an efficient SAT solve. In: Proceedings of DAC 2001, pp. 530–535 (2001)
13. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation 2(3), 293–304 (1986)
14. Prestwich, S.: Local Search on SAT-Encoded Colouring Problems. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 26–29. Springer, Heidelberg (2005)
15. Sakallah, K.A.: Symmetry and Satisfiability. In: Handbook of Satisfiability, ch. 10, pp. 289–338 (2009)
16. Sinz, C., Biere, A.: Extended Resolution Proofs for Conjoining BDDs. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 600–611. Springer, Heidelberg (2006)
17. Tseitin, G.: On the complexity of derivation in propositional calculus. In: Studies in Mathematics and Mathematical Logic, Part II, pp. 115–125 (1968)
18. Urquhart, A.: Hard examples for resolution. Journal of the ACM 34(1), 209–219 (1987)
19. Van Gelder, A.: Another look at graph coloring via propositional satisfiability. Discrete Applied Mathematics 156(2), 230–243 (2008)
20. Zykov, A.A.: On some properties of linear complexes. Amer. Math. Soc. Translations 79, 81 (1952)
21. http://mat.gsia.cmu.edu/COLOR/solvers/trick.c
22. Computational Series: Graph Coloring and Its Generalizations, http://mat.gsia.cmu.edu/COLOR04

Minimizing Learned Clauses

Niklas Sörensson[1] and Armin Biere[2]

[1] Chalmers University of Technology, Göteborg, Sweden
[2] Johannes Kepler University, Linz, Austria

Abstract. Minimizing learned clauses is an effective technique to reduce memory usage and also speed up solving time. It has been implemented in MiniSat since 2005 and is now adopted by most modern SAT solvers in academia, even though it has not been described in the literature properly yet. With this paper we intend to close this gap and also provide a thorough experimental analysis of it's effectiveness for the first time.

1 Introduction

Learning clauses [9] is an essential part in modern SAT solvers [6,8,10]. Learning is used for forward pruning search space and in combination with a conflict-driven assignment loop [8,9] also allows to skip redundant decisions during backtracking. The 1-UIP learning scheme [9] is considered to work best [4,18]. It is possible to increase the efficiency of the 1-UIP scheme, by removing additional literals from learned clauses. This can either be done *locally* [2] or *recursively* [11,16]. The latter was first implemented in MiniSAT in 2005 but has not been properly described in the literature yet.

Learning is usually explained with the help of implication graphs [8], which has assigned variables as nodes connected through *antecedents* [8]. The analysis starts with a clause in which all literals are assigned to false, formally the antecedent of the conflict κ [8], and resolves in antecedents from its *implied* [8] variables recursively. Several termination conditions are possible [18]. In the simplest scheme the process continues until only decisions are left. The standard algorithm [9] makes sure that learned clauses derived this way contain exactly one literal from the current decisions level. This is usually referred to as that the learned clause must be *asserting*.

If antecedents are resolved in reverse assignment order, the first derived asserting clause, is called the *first unique implication point* (1-UIP) clause [8,9]. Learning 1-UIP clauses is considered to be the best learning scheme [4,18]. In extensions [4,7,9,13] additional learned clauses are not asserting, and are added as complement to the 1-UIP clause. However, a proper subset of the 1-UIP clause will necessarily be more successful at pruning future search. This is our original motivation for the algorithms in this paper.

The 1-UIP clause is *minimized* by resolving more antecedents without adding literals. A similar idea appears in [2], which we call *local minimization*. A general version was discovered independently by the first author and implemented in MiniSat 1.13 [16]. This *recursive minimization* is now part of many SAT solvers.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 237–243, 2009.

2 Minimization

The example in Fig. 1 applies the original 1-UIP scheme and the decision scheme. It also explains how the 1-UIP clause can be minimized locally or recursively. More precisely, the 1-UIP clause can be *minimized locally* by resolving out a literal, which has other literals in its antecedent already in the 1-UIP clause. This gives a first version of an algorithm for locally minimizing learned clauses:

> *Generate the 1-UIP clause. Apply self-subsuming resolution, in reverse assignment order, using antecedent clauses for self-subsuming resolution.*

This algorithm actually produces a regular and linear resolution derivation of the minimized clause. In general, resolving antecedents can not introduce cycles, even if resolved out-of-order with respect to assignment order, in contrast to [1]. Furthermore, tree-like resolution can be made regular [17]. Thus literals can actually be deleted in an arbitrary order. This simplifies implementation considerably and results in the following modified algorithm for local minimization:

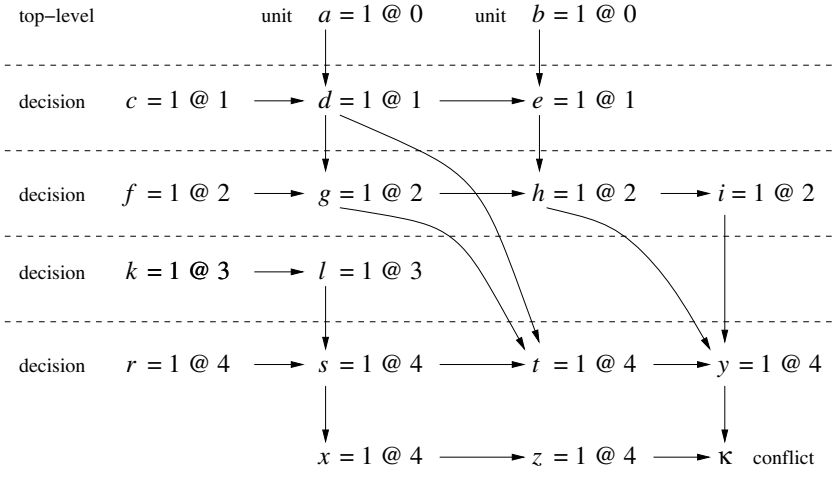


Fig. 1. An implication graph with two top-level unit clauses, and four decisions. The 1-UIP scheme, resolves from the antecedent $(\overline{y} \vee \overline{z})$ of the conflict as few as possible literals until exactly one literal from the current decision level, the 1-UIP $\overline{s}$, is left. The resulting 1-UIP clause is $(\overline{d} \vee \overline{g} \vee \overline{h} \vee \overline{i} \vee \overline{s})$. Depending on the definition, e.g. whether all literals in the derived clause are decisions, or just the UIP on the current decision level, the decision UIP clause either is comprised of the negations of all the decisions $(\overline{c} \vee \overline{f} \vee \overline{k} \vee \overline{r})$ or is obtained from the 1-UIP clause, replacing $\overline{s}$ by $\overline{l} \vee \overline{r}$. In any case, all four non top-level decision levels are "pulled" into the decision UIP clause, while the 1-UIP clause allows to jump over the decision level of k. Local minimization of the 1-UIP clause removes $\overline{i}$, since its single antecedent literal $\overline{h}$ already occurs in the 1-UIP clause. This is an instance of self-subsuming resolution, resolving $(\overline{d} \vee \overline{g} \vee \overline{h} \vee \overline{i} \vee \overline{s})$ with $(\overline{h} \vee i)$ to obtain $(\overline{d} \vee \overline{g} \vee \overline{h} \vee \overline{s})$. No other local minimization is possible. Top-level assigned literals can be ignored. Thus the nodes d and g together dominate h. As a consequence $\overline{h}$ can be deleted using recursive minimization to obtain $(\overline{d} \vee \overline{g} \vee \overline{s})$.

> *Generate the 1-UIP clause. Mark all its literals. Remove those implied variables which have all their antecedent literals marked.*

On the current decision level, enforcing traversal in assignment order presents no overhead, since literals of the current decision level have to be unassigned anyhow. Traversing the *trail* [6] backward gives the desired topological ordering. Traversing all literals on the trail of previous decision levels is more costly.

It is possible to continue resolving out literals, as long all newly introduced literals are *eventually* resolved out. A literal can be deleted if its antecedent literals are, in the implication graph, dominated by other literals from the 1-UIP clause. The recursive minimization algorithm can be formulated as follows:

> *Generate the 1-UIP clause. Mark its literals. Implied variables in 1-UIP clause are candidates for removal. Search implication graph. Start from antecedent literals of candidate. Stop at marked literals or decisions. If search always ends at marked literals then the candidate can be removed.*

Soundness can be proven by simulating graph traversal with resolution. The only issue is, if literals are resolved out, not respecting the reverse chronological assignment order. Again these irregularities can be eliminated by reorganizing the derivation [17]. The result is a regular tree-shaped resolution derivation.

As optimizations successful removals should be cached and we can terminate the search through the implication graph early as soon as a literal from a decision level that is not present in the 1-UIP clause is encountered. This early termination condition can be implemented by marking decision levels of the 1-UIP clause, if decision levels are represented explicitly in the SAT solver, or as in MiniSat and PicoSat, by an over-approximation technique based on signatures as in subsumption algorithms [5].

3 Experiments

To empirically compare the effectiveness of recursive versus local minimization versus no minimization of learned clauses at all, we used the same set of 100 benchmarks as in the SAT Race'08 [14]. The run times were obtained on our 15 node cluster with Pentium 4 CPUs running at 2.8 GHz with 2 GB main memory. The space limit was 1.5 GB and the time limit 1800 seconds.

In order to obtain a statistically valid evaluation, we first employed two different SAT solvers. Additionally we used two versions of each SAT solver, one with preprocessing enabled and one version in which it was disabled. Second we use all 100 SAT Race'08 instances. Third, we injected noise using a random number generator to influence decision heuristics. The same set of benchmarks was then run 10 times with different seeds for each of the four configurations, i.e. one out of two SAT solvers with and without preprocessing. Altogether we have 4000 runs for each of the three variants of the minimization algorithm. The worst case accumulated execution time would have been $3 \cdot 4000 \cdot 1800$ seconds $= 6000$ hours. Since many instances finished before the time limit was reached it actually only took 2513 hours of compute time to finish all 12000 runs.

Table 1. Experiments with MiniSat and PicoSat on SAT Race'08 benchmarks. The first column specifies the configuration, e.g. which of the two SAT solvers is used and whether preprocessing is enabled or disabled. The last three rows summarize these four configurations. The next column specifies the minimization algorithm: "recur" is recursive minimization, the default in MiniSat and PicoSat. Then there are rows with "local" minimization for each configuration and "none" denotes the base case, in which learned clauses are not minimized at all. The third column gives the number of solved instances out of 1000, respectively 4000 in the last three rows. Each row corresponds to the 10 runs with different seeds over the 100 instances. The next column gives the improvement in number of solved instances with respect to the base case. The number of solved instances increases by roughly 10% for recursive minimization, and half that much for local minimization only. The difference in run-time, shown in the next two columns, gives a similar picture. The percentage in the 6th column is calculated as the amount of time the minimizing algorithm finishes earlier relative to the base case. An unsolved instance contributes 1800 seconds. In the next two columns we report on memory usage, calculated as the sum of the maximum main memory used in each run, at most 1.5 GB per run. Half of the memory can be saved using recursive minimization, with local minimization one quarter. This effect is even more dramatic with respect to the number of times a run reached the space limit, which is shown in columns 9 and 10 next, particularly in the case of MiniSat. PicoSat uses more compact data structures than MiniSat, for instance to store binary clauses [12]. Minimization also reduced the number of space-outs by more than 60%. Finally, the last column, shows the average number of deleted literals per learned clause. This is calculated with respect to the size of the 1-UIP clause, which would have been generated without minimization, even though the 1-UIP clause is minimized afterwards. This is different from comparing the average length of learned clauses with and without minimization, since these statistics are computed within the minimizing solver. This gives an explanation why memory savings are almost twice as much as savings due to deleted literals only. Minimization not only saves space, but also reduces the search space.

		solved instances		time in hours		space in GB		out of memory		deleted literals
MiniSat	recur	788	9%	170	11%	198	49%	11	89%	33%
with	local	774	7%	177	8%	298	24%	72	30%	16%
preprocessing	none	726		192		392		103		
MiniSat	recur	705	13%	222	8%	232	59%	11	94%	37%
without	local	642	3%	237	2%	429	24%	145	26%	15%
preprocessing	none	623		242		565		196		
PicoSat	recur	767	10%	182	13%	144	45%	10	60%	31%
with	local	745	6%	190	9%	188	29%	10	60%	15%
preprocessing	none	700		209		263		25		
PicoSat	recur	690	6%	221	8%	105	63%	10	68%	33%
without	local	679	5%	230	5%	194	31%	10	68%	14%
preprocessing	none	649		241		281		31		
	recur	2950	9%	795	10%	679	55%	42	88%	34%
altogether	local	2840	5%	834	6%	1109	26%	237	33%	15%
	none	2698		884		1501		355		

Table 2. Even for 100 benchmarks there is a great variance for different seeds. The columns are as in Tab. 1. Even though the space reduction and also the percentage of deleted literals is consistent for different seeds, the run-time and the number of solved instances vary widely. Both, for MiniSat and PicoSat, it would be possible to draw the conclusion that local minimization is better than recursive minimization, by picking two specific seeds, for instance MiniSat/local/0 vs. MiniSat/recur/2, and PicoSat/local/7 vs. PicoSat/recur/1. The relative space usage is consistent over different runs of the same algorithm, as is the percentage of deleted literals. For empirical evaluations of heuristics of SAT solvers, we suggest, to enforce identical search behavior [3], or to use a very large set of benchmarks, definitely more than 100. However, it is probably necessary to randomize the algorithms and run them with different seeds a sufficient number of times. Another option is to use secondary statistics directly related to the proposed heuristics, like the number of deleted literals in our case, in addition to the number of solved instances, time usage, or a scatter plot.

	MiniSat with preprocessing							PicoSat with preprocessing						
	seed	solved	time	space	mo	del		seed	solved	time	space	mo	del
recur	8	82	16	19	1	33%	recur	9	79	17	14	1	31%
recur	6	81	17	20	1	33%	recur	0	78	18	14	1	31%
local	0	81	16	29	7	16%	recur	3	78	18	14	1	31%
local	7	80	17	29	8	15%	recur	8	78	18	14	1	31%
recur	4	80	17	20	1	33%	recur	2	77	19	14	1	31%
recur	1	79	17	20	1	33%	local	7	77	19	19	1	15%
recur	9	79	17	20	1	34%	recur	6	77	18	14	1	31%
local	5	78	18	29	7	16%	local	3	77	18	18	1	15%
local	1	78	17	29	6	16%	recur	7	76	18	14	1	31%
recur	0	78	17	20	1	34%	local	4	75	19	19	1	15%
recur	5	78	17	19	1	33%	local	1	75	19	19	1	15%
local	3	77	18	31	7	16%	recur	4	75	18	14	1	31%
local	8	77	18	30	8	16%	recur	5	75	18	14	1	30%
recur	7	77	17	20	1	34%	local	2	74	19	19	1	15%
recur	3	77	17	20	1	34%	local	8	74	19	19	1	15%
recur	2	77	17	20	2	33%	recur	1	74	19	14	1	31%
none	7	76	19	39	9	0%	local	5	74	19	18	1	15%
local	2	76	18	31	8	16%	local	6	73	20	19	1	15%
local	4	76	18	31	7	16%	local	0	73	20	19	1	15%
local	6	76	18	30	7	16%	local	9	73	19	19	1	16%
local	9	75	19	29	7	16%	none	5	72	21	26	4	0%
none	9	74	19	39	10	0%	none	3	72	20	26	3	0%
none	6	73	19	40	12	0%	none	7	72	20	26	2	0%
none	3	73	19	39	10	0%	none	8	71	21	27	2	0%
none	8	72	20	39	11	0%	none	9	71	20	25	3	0%
none	0	72	20	39	11	0%	none	1	70	21	27	1	0%
none	1	72	19	39	9	0%	none	4	69	21	26	2	0%
none	5	72	19	39	10	0%	none	0	69	21	26	4	0%
none	2	71	20	40	11	0%	none	6	68	21	26	2	0%
none	4	71	19	39	10	0%	none	2	66	22	27	2	0%

As first SAT solver we use an internal version of MiniSat [6], a snapshot from November 11, 2008. It is almost identical to the one described in [15] the winner of the SAT Race'08. It additionally allows to perturbate the initial variable ordering slightly using a pseudo random number generator. The second SAT solver is PicoSat [3] version 880, an improved version of PicoSat [3]. The major improvement was to separate garbage collection of learned clauses from restart scheduling. One out of 1000 decisions is a random decision in PicoSat. The seed for the random number generator is specified on the command line. Table 1 shows our main experimental results. In Tab. 2, we focus on two configurations. More details can be found at `http://fmv.jku.at/papers/minimize.7z`.

4 Conclusion

In this paper we discussed algorithms for minimizing learned clauses. Our extensive experimental analysis proves the effectiveness of clause minimization.

References

1. Audemard, G., Bordeaux, L., Hamadi, Y., Jabbour, S., Sais, L.: A generalized framework for conflict analysis. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 21–27. Springer, Heidelberg (2008)
2. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. J. Artif. Intell. Res. (JAIR) 22 (2004)
3. Biere, A.: PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation (JSAT) 4 (2008)
4. Dershowitz, N., Hanna, Z., Nadel, A.: Towards a better understanding of the functionality of a conflict-driven SAT solver. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 287–293. Springer, Heidelberg (2007)
5. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
7. Jin, H.S., Somenzi, F.: Strong conflict analysis for propositional satisfiability. In: Proc. DATE 2006 (2006)
8. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, IOS Press, Amsterdam (2009)
9. Marques-Silva, J., Sakallah, K.: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Trans. on Computers 48(5) (1999)
10. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC 2001 (2001)
11. Nadel, A.: Understanding and Improving a Modern SAT Solver. PhD thesis, Tel Aviv University (submitted, 2008)
12. Pilarski, S., Hu, G.: Speeding up SAT for EDA. In: Proc. DATE 2002 (2002)
13. Pipatsrisawat, K., Darwiche, A.: A new clause learning scheme for efficient unsatisfiability proofs. In: Proc. AAAI 2008 (2008)
14. Sinz, C.: SAT-Race 2008 (2008), `http://baldur.iti.uka.de/sat-race-2008`

15. Sörensson, N., Eén, N.: MS 2.1 and MS++ 1.0, SAT Race 2008 edn. (2008)
16. Sörensson, N., Eén, N.: MiniSat v1.13 – A SAT solver with conflict-clause minimization (2005)
17. Urquhart, A.: The complexity of propositional proofs. Bull. of EATCS 64 (1998)
18. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: Proc. ICCAD 2001 (2001)

Extending SAT Solvers to Cryptographic Problems

Mate Soos[1], Karsten Nohl[2], and Claude Castelluccia[1]

[1] INRIA Rhône-Alpes
[2] University of Virginia

Abstract. Cryptography ensures the confidentiality and authenticity of information but often relies on unproven assumptions. SAT solvers are a powerful tool to test the hardness of certain problems and have successfully been used to test hardness assumptions. This paper extends a SAT solver to efficiently work on cryptographic problems. The paper further illustrates how SAT solvers process cryptographic functions using automatically generated visualizations, introduces techniques for simplifying the solving process by modifying cipher representations, and demonstrates the feasibility of the approach by solving three stream ciphers.

To optimize a SAT solver for cryptographic problems, we extended the solver's input language to support the XOR operation that is common in cryptography. To better understand the inner workings of the adapted solver and to identify bottlenecks, we visualize its execution. Finally, to improve the solving time significantly, we remove these bottlenecks by altering the function representation and by pre-parsing the resulting system of equations.

The main contribution of this paper is a new approach to solving cryptographic problems by adapting both the problem description and the solver synchronously instead of tweaking just one of them. Using these techniques, we were able to solve a well-researched stream cipher 2^6 times faster than was previously possible.

1 Introduction

Cryptographic functions are at the base of computer security with encryption ciphers ensuring confidentiality and authenticity. Despite their importance, many practical cryptographic functions rely on unproven assumptions about the complexity of their underlying mathematical problems. When these assumptions are found to be incorrect, new theoretical and practical attacks are constructed that sharpen the understanding of a specific problem and advance the evolution of cryptography in general. SAT solvers have been shown to be a powerful tool in testing mathematical assumptions. In this paper, we extend SAT solvers to better work in the environment of cryptography.

Previous work on solving cryptographic problems with SAT solvers has concentrated on the best mathematical representation of ciphers [1]. To further

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 244–257, 2009.

improve the potential of SAT solvers, we adapted a SAT solver to better suit cryptographic problems and then manipulated the representation of some cryptographic problems to best fit this modified solver. We refined SAT solvers to understand the XOR operation, which is common in cryptography, besides functions in the conjunctive normal form (CNF) that is native to many SAT solvers. We further added dynamic behavior analysis to more thoroughly understand the workings of SAT solvers on cryptographic primitives.

To show the effectiveness of our approach, we solved a few different ciphers. The first two targets, Crypto-1 [2] and HiTag2 [3], are weak stream ciphers, widely used in electronic payment, access control and car immobilizers. Our third target, Bivium [4], is a simplified version of the eSTREAM standard stream cipher Trivium [5] known for its simple description. Solving these ciphers with an unmodified SAT solver and with only basic improvements to their CNF representation reveals the secret state within 170 hours for Crypto-1, a week for HiTag2 and in $2^{42.7}$ s [6] for Bivium. With our adapted SAT solver and tuned cipher description techniques, the average attack time on a desktop PC drops to 40 seconds for Crypto-1, 6.5 hours for HiTag2 and $2^{36.5}$ s for Bivium.

Contributions

We optimize a standard SAT solver for cryptographic problems in Sect. 3. The SAT solver now handles XOR operations natively to faster solve cryptographic problems and the solver's execution is visualized to allow insight into its inner workings. Based on these improvements, guidelines are derived on how to convert ciphers to a description that can be quickly solved in Sect. 4. Finally, three ciphers are solved using the adapted SAT solver faster than was previously possible with other SAT solver-based techniques, in Sect. 5.

2 Background

Our results build on research in stream ciphers, SAT solvers, and algebraic cryptanalysis. This section presents the current state of research in these areas and indicates where they connect.

2.1 Stream Ciphers

A stream cipher is a symmetric cryptographic function that allows two parties to communicate privately when they share a secret key. Stream ciphers produce a stream of pseudorandom bits (the *keystream*) given a secret *key* and a non-secret random initialization vector (IV). This key stream is XORed with a message prior to sending and again XORed after receiving so that the message cannot be read while in transit.

The stream ciphers discussed in this paper are based on one or more shift registers with linear or non-linear feedback function as well as a filter function that maps the register states to keystream bits. Stream ciphers have two phases:

an *initialization phase* followed by a *keystream generation phase*. During initialization, key and IV are typically mixed to become the initial state by shifting the registers while feeding in a combination of the feedback function and the filter function. During keystream generation, the registers are shifted and their feedback function is applied, while the keystream is generated from the state using the filter function.

2.2 SAT Solvers

Satisfiability solvers are programs that employ highly optimized mathematical algorithms to decide whether a set of constraints have a solution. This paper only discusses one widely-used constraint set, the conjunctive normal form (CNF). In CNF, each element in the constraints, a or $\bar{a}$, is called a literal. A clause is a disjunction (`or`-ing) of literals. CNF is a conjunction (`and`-ing) of clauses. Hence, the constraints are presented to the SAT solver as an "and of ors".

SAT solvers are mostly used in electronic design automation (EDA), though they are also used in a growing number of other domains. State-of-the-art solvers have been extended or adopted to meet the specific characteristics of different problem domains, for example temporal induction in [7].

Modern SAT solvers that are based on the DPLL algorithm [8] evolved from GRASP [9] which introduced learning, and later from from Chaff [10] which introduced watched literals and dynamic variable ordering. Solvers that employ these techniques are called *conflict-driven* SAT solvers. In this paper we extend MiniSat [11], a conflict-driven SAT solver designed for researchers to adapt it to different domains.

MiniSat employs a backtracking-based, depth-first search algorithm to find a satisfying variable assignment for a system of clauses. The algorithm branches on a variable by guessing it to `true` or `false` and examining whether the value of other variables depends on this guess. The affected variables are then assigned and the search continues until no more assignments can be made. During this period, called *propagation*, a clause may be found that cannot be satisfied anymore. If such a *conflict* is encountered, a *learned clause* is generated that captures the wrong guesses that lead to the conflict. The topmost guess allowed by the learned clause is then reversed and the algorithm continues. The learned clauses trim the search tree and guide the algorithm in choosing the best next guess. Eventually, either a satisfiable assignment is found or the search tree is exhausted, meaning that no solution exists.

2.3 Algebraic Cryptanalysis

Algebraic cryptanalysis is a family of attacks that exploits insufficient complexity in ciphers. These attacks have successfully been applied to break a number of ciphers secure against other forms of cryptanalysis. In algebraic attacks, equations are constructed that express the output bits of a cipher in terms of its inputs, or its state. These equations are then solved and reasoned about with either dedicated equation solvers such as the F5 algorithm [12], or standard SAT solvers.

The first SAT-based cryptanalysis was by Massacci et al. [13], experimenting with the Data Encryption Standard (DES) using DPLL-based SAT solvers. More recent work by Courtois and Bard has produced attacks against KeeLoq [1] and stream ciphers with linear feedback [14]. Algebraic cryptanalysis has also been used on modern stream ciphers, such as the reduced version of Trivium, Bivium [4].

3 Adapting the SAT Solver

To take full advantage of the power of SAT solving we adapted and optimized MiniSat, a state-of-the-art DPLL-based SAT solver, for algebraic cryptanalysis. We further added visualization to the solver to help identify bottlenecks and improve the solving by altering the problem representation. Among the many choices for modern SAT solvers, we chose MiniSat for its competitive performance, code availability, and a design that specifically encourages extensions to its input language.

3.1 XOR Support

Cryptographic building blocks such as filter and feedback functions lead to equations with many XORs. These XOR constraints, when converted to CNF representation without further elaboration, grow exponentially in size. This is because the XOR constraint's Karnaugh table contains 2^{len-1} minterms, and hence needs 2^{len-1} clauses to describe in CNF.

To circumvent this limitation, previous research extended the Satz solver to reason about 2- and 3-long XOR constraints, which they called equivalency reasoning [15]. For MiniSat, previous research [1, Sect. 6.4] cut up the XOR function into groups of smaller XORs, each setting an additional variable. The full XOR was then represented as a XOR of the additional variables.

While cutting up XORs allows MiniSat to work on long XOR chains, this approach forces the solver to watch and examine many clauses for variable changes, when in fact only one XOR constraint should be watched. To mitigate this limitation, we implemented the XOR constraint natively into MiniSat. Each XOR constraint is represented by a single *xor-clause*. A xor-clause behaves as a regular clause towards all unchanged parts of the solver: it dynamically changes appearance when propagating or causing a conflict by appearing as a different regular clause depending on the current assignment of variables.

For example, the xor-clause $a \oplus b \oplus c$ represents all the regular clauses

$$a \vee \overline{b} \vee \overline{c} \quad (1) \qquad \overline{a} \vee \overline{b} \vee c \quad (2)$$
$$a \vee b \vee c \quad (3) \qquad \overline{a} \vee b \vee \overline{c} \quad (4)$$

and if, for example $a = \texttt{true}$ and $b = \texttt{true}$, then it changes its appearance to the regular clause (2), and causes the propagation $c = \texttt{true}$ just as its regular representation would. If, however, $a = \texttt{false}$, $b = \texttt{true}$ and $c = \texttt{true}$, the xor-clause changes its appearance to regular clause (1) and causes a conflict just as its regular representation would.

Generating a conflicting or propagating clause from a xor-clause is done as follows. All variables that are assigned to `false` are included as-is, and all variables that are assigned to `true` are included in a negated form. If propagating, the single unassigned variable is also included, its negation depending on the values of the other variables in the xor-clause.

Solving cryptographic functions is accelerated considerably by integrating xor-clauses into MiniSat. For the stream ciphers Crypto-1 and Grain solving is up to twice as fast with xor-clauses and memory usage is decreased by at least an order of magnitude.

Besides speeding up the solving, native XOR support leads to more concise input file and internal data structures, which simplify analyzing the dynamic behavior of the solver. Lastly, xor-clauses enable a straightforward implementation of Gaussian elimination into MiniSat as explained in the next section.

3.2 Gaussian Elimination

Gaussian elimination is an efficient algorithm for solving systems of linear equations. Since each xor-clause is a linear equation, we can use this algorithm to solve the system of equations described by the xor-clauses. Some linear problems with as many as 100 variables can be trivially solved with Gaussian elimination but take an excessive amount of time when solved with SAT solvers. This phenomenon is due to the fact that SAT solvers solve by guessing variables and determining if there is any equation that gives a result given the current assignments. If the set of linear equations is dense (i.e. all equations contain many variables), almost all variables need to be guessed before any equation gives a result. Thus, for a system with 100 variables, it is not uncommon that 80 variables need to be guessed before any equation gives a result, i.e. the search space is on the order of 2^{80}. When using Gaussian elimination, on the other hand, the same problem can be solved in less than 2^{20} operations.

Since Gaussian elimination and the DPLL algorithm (used in MiniSat) are optimal for different parts of cryptographic problems, the best results are achieved by switching between the two. To benefit from Gaussian elimination during solving, whenever the SAT solver cannot perform any further propagations and would need to guess a variable, we ask the Gaussian elimination if there is any information it could extract from the xor-clauses.

Execution of the Gaussian elimination gives one of the following results: either it finds nothing, or it finds that a variable can be propagated by a combination of xor-clauses, or it finds that given the current assignments, the system of equations is unsatisfiable. In the two latter cases, the solver needs the actual xor-clause, which when evaluated with the variable assignments, gives a unit or empty clause, respectively. This actual xor-clause is important, as it signals the solver what variable was propagated by what clause (in case of a propagation), or what clause caused the conflict (in case of a conflict). To calculate the actual xor-clauses, we keep two matrixes: one updated with the current assignments, and one that mirrors the other only with its row-swap and row-xor operations. Whenever there is a propagation or conflict indicated by the first matrix, the

second matrix is used to generate the actual xor-clause. For example, if the two matrixes are:

xor-clauses with $v8$ assigned to `true`					actual xor-clauses				
$v10$	$v8$	$v9$	$v12$	aug	$v10$	$v8$	$v9$	$v12$	const
1	–	1	1	1	1	1	1	1	0
0	–	1	1	1	0	0	1	1	1
0	–	0	1	0	0	1	0	1	1
0	–	0	0	0	0	1	0	0	1

then the second to last row of the first matrix indicates propagation of $v12 =$ `false`. The actual xor-clause can be read from the second matrix: it is $v8 \oplus v12 \oplus 1$. The matrix used by the Gaussian elimination algorithm is upper triangular, but the matrix containing the actual xor-clauses is only upper triangular for the columns representing variables that are not assigned.

Including Gaussian elimination into MiniSat is based on the idea of *SAT Modulo Theories* (SMT). An SMT instance is a generalization of a Boolean SAT instance in which various sets of variables are replaced by predicates from a variety of underlying theories. Naturally, SMT formulas provide a much richer modeling language than is possible with Boolean SAT formulas. In essence, xor-clauses enrich MiniSat's language, which the Gaussian elimination can understand and reason about, tightly integrating its conclusions into the DPLL algorithm of MiniSat.

A trade-off parameter for the Gaussian elimination is the cut-off depth until which it is worthwhile to execute the algorithm. Cutting off branches at the top reduces the search space more than cutting at the bottom, but it takes approximately the same time to execute the algorithm. However, if the cut-off depth is too shallow, the constant overhead is more than the benefit, but if too deep, the dynamic overhead is more than the benefit. In the end, we made the cut-off depth configurable, and ran tests to decide for each cipher which depth gave the most benefit.

To save time, the matrix is incrementally normalized as the solver travels down the search tree and assignments are made. We save the matrix at every search depth, and in case the solver has to jump back (due to a conflict), we re-load the matrix from the state saved at that depth.

Using Gaussian elimination, solving Bivium and Trivium is faster by 1-5% if we restrict the search depth to between 1 and 8, depending on the number of guessed bits. For other instances derived from other ciphers, Gaussian elimination does not appear to decrease the overall solving time. A comparative figure for Bivium, showing the speed of solving and the explored search space versus the depth until which the algorithm was active is present in Fig. 1. It is apparent from the graphs that using Gaussian elimination reduces the explored search space (in the example, by up to 83%), but the algorithm takes more and more time to execute as the cut-off depth is increased.

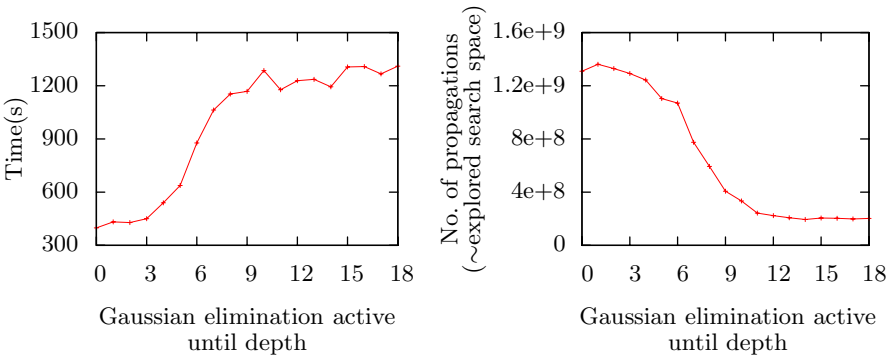


Fig. 1. Comparison between the time and the number of propagations (∼explored search space), relative to the depth until which the Gaussian elimination was active. Each point in the graphs represent 2000 random examples of the Bivium cipher, given 56 randomly guessed state bits.

Apart from the marginal speedup that Gaussian elimination brings, it is a useful tool for multiple other reasons. First of all, it demonstrates that SAT solvers ignore certain characteristics of the problem they are dealing with, and by exploiting these properties the search space could be significantly reduced. Secondly, the combined solver works much faster on problems that have large parts that can benefit from Gaussian elimination. Lastly, our implementation of Gaussian elimination can likely be improved upon leading to greater speedups.

3.3 Dynamic Behavior Analysis

The dynamic behavior of SAT solvers is hard to follow since branching and propagation occur far too many times to be traceable by hand. Understanding the solver's dynamic behavior, however, is essential for estimating a cipher's complexity and for improving the solver's performance.

To better understand how MiniSat reaches solutions, we implemented search tree tracing into the solver. The output of our MiniSat trace extension can be analyzed visually and statistically. Visualizing the operation of DPLL-based SAT solvers was introduced in [16], which our implementation augments in multiple ways. Our extension allows for variables to be named and for clauses to be grouped, which is useful when multiple clauses are used to represent one logical entity (e.g. a feedback function). The calculated statistics include the type of most conflicted clauses (e.g. filter functions), the average number of propagations per search tree branch, etc. An example search tree of the Crypto-1 cipher is in Fig. 2. The visualization allowed us, for instance, to identify the regularly placed filter function taps of Crypto-1 as its largest weakness over an improved variant found in HiTag2 tags.

It is clear from the variable branching statistics that during solving, the most important variables are picked automatically by MiniSat, which are always the

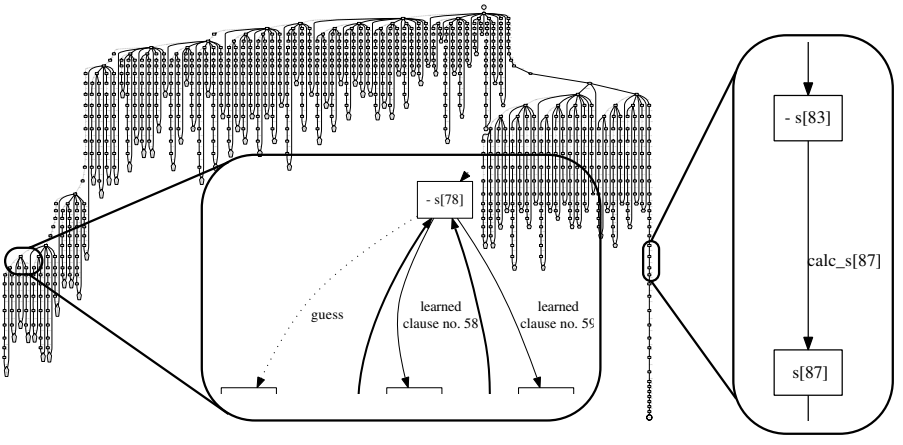


Fig. 2. Graphviz visualization of an example search for the Crypto-1 cipher's states. The tree is read from left to right, top to bottom: the left- and bottommost pentagon is the first conflict clause, the right- and bottommost circle is the satisfying assignment.

state or key bits. By examining smaller search graphs and the statistics on the most conflicted clause groups, we further found that once the important variables have been guessed, the results of these assignments are propagated to the equations representing the known keystream bits, and if they do not match, a conflict occurs, a guess is reversed, and the algorithm starts again.

The solver's strategy is therefore similar to a brute force search in which all key or state bits are tried. If one or more keystream bits can be evaluated without knowing all state bits, the SAT solver will evaluate them, and if the equations do not work out, stop the computation there, effectively doing partial evaluation. Furthermore, clauses are learnt during the search, which later prune the search tree, helping to perform partial evaluation.

The lessons learnt from search-tree tracing are as follows. It is best not to include long initialization sequences (such as that used by Grain) in the equations since after initialization all keystream bits depend on all key bits. This forces the solver to calculate a large part of the cipher in an ineffective way, as its description and subsequent evaluation in the solver is more complicated than the way the cipher was originally meant to be calculated.

A stream cipher is considered broken if its state can be determined at any point during keystream generation. Therefore, instead of making initialization part the problem, its state at a suitable point should be treated as the unknown, as this is the only possible way to take advantage of the partial evaluation property of SAT solvers. Although this state is larger than the key for all modern ciphers, it is relatively easy to solve a large part of it, as the keystream bits depend much more directly on the selected state's bits.

4 Adapting the Cipher Representation

Finding the best representation of stream ciphers in regular and xor-clauses is a crucial step in breaking a cipher with SAT solvers [1, Sect. 8]. For the techniques in this paper, a cipher is described as a logical circuit with functions, variables, the known keystream, and known inputs.

4.1 Logical Circuit Representation

In the logical circuit representation used in our approach, the unknown is the reference state's bits, and functions are expressed in regular and xor-clauses, using variables as input. An example logical circuit for a 3-bit state stream cipher is given in Fig. 3. In the figure, the cipher produces four keystream bits, and the shift register is shifted three times, using the feedback function. Functions are shown as hexagons, variables as simple boxes, and the reference state is marked in gray.

The *depth* of a keystream bit is the number of distinct functions (resp. hexagons) traversed on the way from the keystream bit to the reference state bits. For example, on Fig. 3, the 1st keystream bit's depth is one, while the 4th keystream bit's is four. Since the solver guesses the reference state's bits, the depth of the circuit indicates the number of functions that must be evaluated by the solver to realize that a wrong guess was made for a given keystream bit. Therefore, the shallower the overall depth of the circuit, the faster the solving. The difficulty hidden behind the functions (resp. hexagons) is also relevant, as when traversed, these must be calculated. If the number and length of clauses representing these hexagons are large, the solver is slowed down considerably.

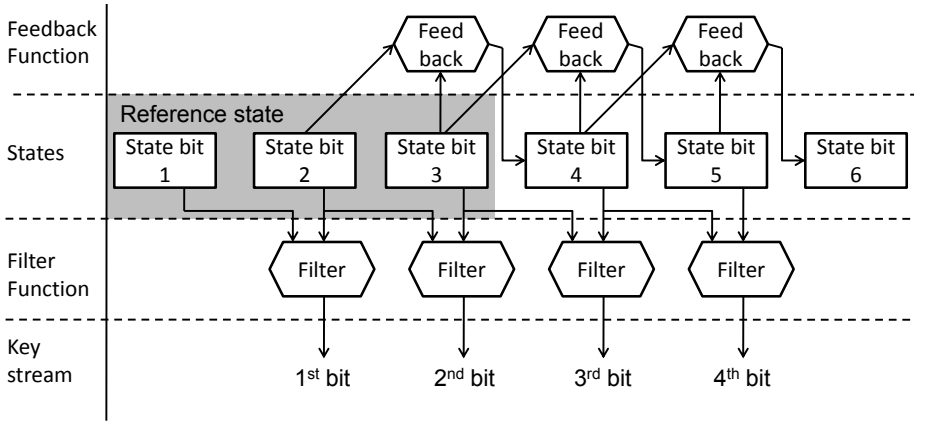


Fig. 3. Logical circuit representation of an example stream cipher: The cipher has a 3-bit shift register, whose filter function depends on the first two bits in the register, and whose feedback function depends on the last two bits in the register

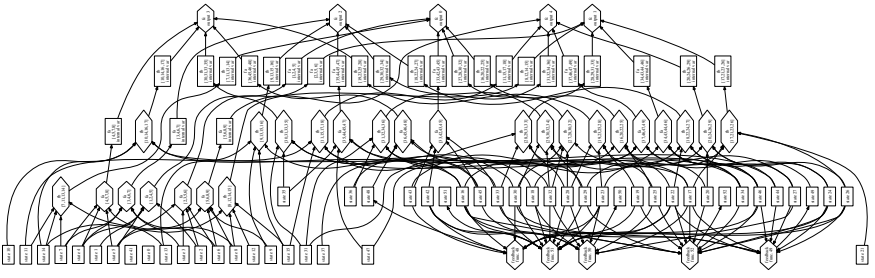


Fig. 4. Clause- and variable-dependency graph of HiTag2. Clause groups are represented as hexagons, and variables as boxes. The known keystream bits are the 5 final filter functions at the top, and the feedback functions are the 5 hexagons at the bottom right.

Finally, the number of reference state bits each keystream bit depends on plays an important role during solving, as a large part of these must be guessed before evaluation can take place. This *dependency number* can be calculated by simply traversing the graph in a breath-first search fashion from the keystream up. The lower this number, the faster the solving.

To summarize, when attempting to represent a cipher, the depth of the resulting logical circuit, the number of reference state bit dependencies and the complexity of the traversed functions' representations must all be optimized to maximize solving speed.

4.2 Generating the Logical Circuit Representation

To evaluate the effectiveness of different representations of the same stream cipher, we extended MiniSat by a tool that generates the logical circuit's description. Given some additional information in the input language, the circuit is visualized with `Graphviz` or statistically analyzed to calculate keystream bit depths and state-bit dependencies. In the generated circuit, just as in the search tree, clauses are grouped into logical elements (such as a filter function), and variables are named (such as reference state bit). An example visualization of HiTag2's logical circuit representation is in Fig. 4.

Having the logical circuit representation allowed us to implement a dependency-tree walker that removes functions whose output does not contribute to any keystream bits, e.g. the last feedback function in Fig. 3. The method used is in essence the same that is used in electric circuit design to remove unused elements, applied to the domain of SAT-based cryptanalysis. Removing useless functions gives only a minor speedup of about 1%, however, unnecessary functions no longer show up on the dynamic behavior analysis statistics, which helps in understanding the inner workings of the solver.

4.3 Optimizing the Representation of LFSRs

Most stream ciphers contain one or more linear feedback shift registers (LFSR). For these ciphers, the state bits not in the reference state can be either be

deduced by continually applying the forward and backward feedback functions or be directly calculated from the reference state's bits. This latter option increases the interdependency of the resulting equations, which helps the solver generate learned clauses that are useful for a larger part of the search tree. These learnt clauses are then used later to avoid useless branches of the search tree, reducing the overall search time.

To generate r keystream bits, r distinct states are needed since generating the n-th keystream bit requires the filter function to be applied to the n-th state. For the solving to be fast, we need to choose the reference state that generates the least complex logical circuit representation. In particular, we must minimize both the average depth and the reference state bit dependencies. According to our experience, this optimal reference state is usually near the $r/2$-th state. As an example, if we had taken state 2 (i.e. state bits 2 to 4) as reference in Fig. 3, the overall depth of the circuit would have been reduced.

4.4 Optimizing the Representation of Non-linear Functions

For efficient solving, the number of clauses, the average clause length, and the number of variables should all be low, but often there exists a trade-off between the three properties.

As an example, the simple $\mathbb{GF}(2)$ polynomial

$$x_1 \oplus x_1x_2 \oplus x_2x_3 \oplus x_1x_3$$

has a Karnaugh table presentation in CNF of

$$\bar{x}_1 \vee \bar{x}_3 \qquad \bar{x}_2 \vee x_3 \qquad x_1 \vee x_2$$

However, the same polynomial can be represented with each non-single monomial expressed as a function, setting additional variables $i_1 \dots i_3$. The polynomial then becomes

$$x_1 \oplus i_1 \oplus i_2 \oplus i_3$$

Using this representation, the number of clauses increases to 3×3 regular $+ 1$ xor-clause, and the average clause length increases to 4.14. Three extra variables also need to be added, diluting the possible learnt clauses with extra variables, thus reducing the effectiveness of learning.

The trade-offs between the two representation methods are complex; from our experience with Grain, Trivium, Crypto-1 and other ciphers, we find that the Karnaugh-table representation works well for functions that contain few (up to 5-6) variables and where these variables are often repeated in many monomials. For instance, solving HiTag2 and Crypto-1 are both sped up by a factor of up to 9x using the Karnaugh table representation.

When a polynomial can be broken up into sub-functions that do not share variables among themselves, such as the polynomials representing the filter functions of Crypto-1 and HiTag2, then these sub-functions must be modelled separately. This increases the overall depth of the resulting logical circuit, however, the complexity of the individual functions traversed during solving is much lower, which is crucial for the solver.

5 Implemented Attacks

The extended SAT solver can solve many stream ciphers. Attacks against three ciphers have been implemented that are faster than any previous SAT solver-based attacks. The first two targets, Crypto-1 and its relative HiTag2, are a weak ciphers used in contactless cards and car immobilizers. The third target, Bivium, is a simplified version of Trivium, a modern cipher standardized through the eSTREAM competition. The solving times for Crypto-1, HiTag2, and Bivium are in Table 1, and their detailed discussion is below.

5.1 Crypto-1 and HiTag2

The Crypto-1 stream cipher [2] is implemented on the NXP Mifare Classic card, which is widely used for micropayment in public transport and for building access control. The cipher was designed to have a particularly small hardware footprint consisting of an 48-bit LFSR and a network of small binary functions that form the filter function. HiTag2 [3], used in car immobilizers, is a relative of Crypto-1, and shares its structure but uses different feedback and filter functions.

The security of Crypto-1 has already been broken using MiniSat by Courtois et al. [17]. They did not publish the details of their attack but only stated that secret keys can be found within 200 seconds on average on a PC given 56 bits of keystream. Their attack, however, modifies the equations describing the cipher by mathematical means, which makes their techniques mostly orthogonal to ours. With our method, solving Crypto-1 using 56 bits of known keystream takes 40 s, while solving HiTag2 given the same number of keystream bits takes $2^{14.5}$ s.

5.2 Bivium

The Bivium stream cipher [4], is a reduced version of the original Trivium cipher, and is intended to be used solely as a research tool to analyze the original cipher. The papers that have been published on this cipher [4,6] improve on each other's results, the best of which is solving in $2^{42.7}$ s on a desktop machine.

Bivium can be solved by describing it in MiniSat using the enhancements and insight presented in this paper. To let the solver finish within reasonable time, we randomly guessed some randomly picked reference state bits and did a thousand different runs for each configuration. With this approach the time to solve is exponential in the number of guessed state bits, as illustrated in Fig. 5. Due to

Table 1. Running times for solving Crypto-1, HiTag2, and Bivium

	Vanilla MiniSat	Karnaugh optimization	Karnaugh and xor-clause optimizations
Crypto-1	500 s	72 s	40 s
HiTag2	$2^{17.8}$ *s*	2^{15} s	$2^{14.5}$ s
Bivium	$2^{36.7}$ s	$2^{36.7}$ s	$2^{36.5}$ s

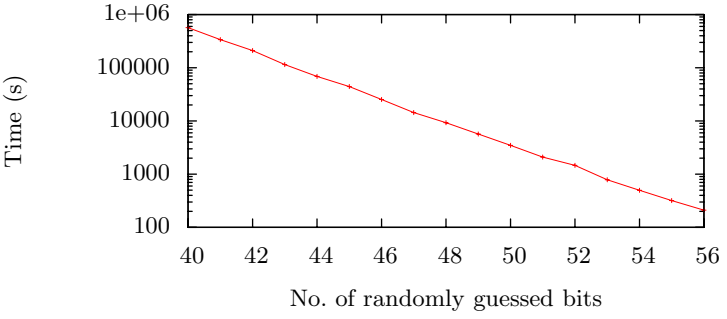


Fig. 5. Solving the Bivium cipher 1000 times, with randomly selected and assigned guess bits. The time to solve is exponential in the number of guess bits.

the large amount of random runs for each point, we can safely extrapolate the graph, giving the result that solving Bivium's state given 177 keystream bits takes about $2^{36.5}$ s.

To generate this result, Gaussian elimination was turned off, as it proved to slow down the solver if less than 58 reference state bits were guessed – for more than 58 guessed bits however, Gaussian elimination with cut-off depth 8 gave an average 5% speedup.

6 Conclusions

SAT solvers are a powerful tool in the analysis of mathematical assumptions, including cryptographic hardness and complexity assumptions. The full potential of SAT solving can only be achieved by matching the problem description to the solver language. For cryptographic ciphers, matching the solver and the problem requires extensive changes to the solver itself. We implemented several steps towards a specialized SAT solver for cryptography including native support for the XOR operation, Gaussian elimination, and logical circuit generation.

The extended solver solves problems from its target domain, simple and complex stream ciphers, faster than any other known SAT-solver based techniques. The Crypto-1 cipher is solved in 40 seconds, HiTag2 in $2^{14.5}$ s, while Bivium takes $2^{36.5}$ s, 2^6 times less than the previous best SAT solver-based attack [6]. Stream ciphers can be strengthened against the attacks presented in this paper through the use of larger states, more complex feedback functions, and through longer initialization phases.

References

1. Bard, G.V.: Algorithms for the solution of polynomial and linear systems of equations over finite fields, with an application to the cryptanalysis of KeeLoq. Technical report, University of Maryland Dissertation (April 2008)

2. Garcia, F.D., et al.: Dismantling MIFARE Classic. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 97–114. Springer, Heidelberg (2008)
3. Nohl, K.: Description of HiTag2 (Webpage), http://cryptolib.com/ciphers/hitag2/
4. Raddum, H.: Cryptanalytic results on Trivium. Technical Report 2006/039, ECRYPT Stream Cipher Project (2006)
5. De Cannière, C.: Trivium: A stream cipher construction inspired by block cipher design principles. In: Katsikas, S.K., López, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) ISC 2006. LNCS, vol. 4176, pp. 171–186. Springer, Heidelberg (2006)
6. McDonald, C., Charnes, C., Pieprzyk, J.: Attacking Bivium with Minisat. Technical Report 2007/040, ECRYPT Stream Cipher Project (2007)
7. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. In: Proc. of Intl. Workshop on Bounded Model Checking. ENTCS, vol. 89 (2003)
8. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (1960)
9. Marques, J.P., Karem, S., Sakallah, A.: Conflict analysis in search algorithms for propositional satisfiability. In: Proc. of the IEEE Intl. Conf. on Tools with Artificial Intelligence (1996)
10. Malik, S., Zhao, Y., Madigan, C.F., Zhang, L., Moskewicz, M.W.: Chaff: Engineering an efficient SAT solver. In: Design Automation Conference, pp. 530–535 (2001)
11. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
12. Faugère, J.C.: A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In: ISSAC 2002, pp. 75–83. ACM Press, New York (2002)
13. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT-problem: Encoding and analysis. Journal of Automated Reasoning 24, 165–203 (2000)
14. Courtois, N.T.: Fast algebraic attacks on stream ciphers with linear feedback. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 176–194. Springer, Heidelberg (2003)
15. Li, C.M.: Equivalency reasoning to solve a class of hard SAT problems. Information Processing Letters 75(1-2), 75–81 (1999)
16. Sinz, C.: Visualizing SAT instances and runs of the DPLL algorithm. J. Autom. Reason. 39(2), 219–243 (2007)
17. Courtois, N.T., Nohl, K., O'Neil, S.: Algebraic attacks on the Crypto-1 stream cipher in Mifare Classic and Oyster cards. Technical Report 2008/166, Cryptology ePrint Archive (2008)

Improving Variable Selection Process in Stochastic Local Search for Propositional Satisfiability

Anton Belov and Zbigniew Stachniak*

Department of Computer Science and Engineering,
York University, Toronto, Canada
{antonb,zbigniew}@cse.yorku.ca

Abstract. This paper considers two methods for speeding-up stochastic local search SAT procedures. The first method aims at using the search history (represented by additional formulas derived at every state of the search process) to constrain the selection of candidate variables used to navigate through the search space of truth-value assignments. The second method uses the search history to allow multiple modifications of the current truth-value assignment in a single search step. Empirical studies of these two methods have demonstrated their effectiveness on structured and industrial SAT instances.

1 Introduction

This paper introduces two techniques for improving stochastic local search (SLS) for models of Boolean formulas in the space of truth-value assignments. Informally speaking, a SLS solver in search state h that falsifies the input formula α selects and moves to one of the neighboring states of h by changing ('flipping') the truth-value of one of the variables of α from that assigned by h to the opposite one. The first speedup technique proposed in this paper, named *candidate variable trimming procedure*, narrows the choice of the next state by disallowing some of the variables of α from being selected for the flip of their truth-values. It is then the purpose of the variable selection heuristic of the solver to select a variable v that is not blocked by the candidate variable trimming procedure and to form the next state h_v.

The second technique discussed in this paper, named *literal commitment strategy*, refines the way the next state of the search for a model of an input formula is selected. In short, this technique forces a SLS solver to further modify the state h_v described above by flipping the truth-values of some additional variables. These additional variables are determined by reasoning about the search history represented by additional formulas that the solver is required to derive from the input formula at every state of the search.

Both techniques are applicable to clausal as well as non-clausal SLS SAT solvers. When incorporated into the CNF solver UBCSAT ([13]) and into the non-clausal solver polSAT-N ([12]) they markedly improved the performance of the solvers on structured and industrial problems. However, the utility of these speed-up techniques for random SAT instances seems to be less than favorable.

* Research of both authors supported by grants from the Natural Sciences and Engineering Research Council of Canada.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 258–264, 2009.

We assume that the reader is familiar with the fundamental concepts of SLS and complete backtracking SAT solving methods (as discussed, for instance, in [5,6,9,11]).

The entailment operation of classical propositional logic (CPL) is denoted by $\vdash$. Let α be a formula of CPL. $Var(\alpha)$ denotes the set of all the variables that occur in α. A truth-value assignment for α is a function h mapping $Var(\alpha)$ into the set $\{0, 1\}$ of *truth-values*. Since CPL's semantics allows the unique extension of every truth-value assignment h to the set of all sub-formulas of α, we shall be making no distinction between h and its extension. Finally, a truth-value assignment h is a *model* (resp. a *countermodel*) for α, if $h(\alpha) = 1$ (resp. $h(\alpha) = 0$).

2 Selecting Candidate Variables

A generic SLS procedure for SAT is presented in Figure 1. For every input formula α, the domain of a variable selection heuristic is restricted to the set $Cand(\alpha, h)$ of candidate variables computed in line (1) for each value of the current truth-value assignment h. In many clausal SLS systems, such as WalkSAT ([11]), $Cand(\alpha, h)$ is the set of variables that occur in some conjunct of α that is false under h. In the non-clausal solver polSAT ([12]), $Cand(\alpha, h)$ is computed based on the structure of α.

```
procedure SLS-SAT(α)
    for i := 1 to MaxTries do
        h :=random truth-value assignment
        for j := 1 to MaxFlips do
            if h(α) = 1 then return h else
(1)         compute Cand(α, h) ⊆ Var(α);
(2)         pick v ∈ Cand(α, h) using a variable selection heuristic; h(v) := 1 − h(v);
            end if
        end for
    end for
    return 'satisfying valuation for α not found'
```

Fig. 1. Generic SLS procedure for SAT

Henceforth, we shall consider SLS procedures S that satisfy the following properties. Given an input formula α and its countermodel h,

(s1) for every model h' for α, there exists $p \in Cand(\alpha, h)$, such that $h'(p) \neq h(p)$;
(s2) S generates a clause c_h such that $Var(c_h) \subseteq Cand(\alpha, h)$, $h(c_h) = 0$, and $\alpha \vdash c_h$.

Each clause c_h described in (s2) can be viewed as a partial record of the current state of the search as represented by h. A solver can keep track of the search history by storing and maintaining these clauses. Therefore we shall also assume that

(s3) an SLS SAT solver retains clauses c_h in a database DB.

Let α be a satisfiable formula and h be one of its countermodels. The property (s1) guarantees that there is at least one variable in $Cand(\alpha, h)$ whose flip would bring us closer

to a model for α. This property does not ensure, however, that the choice of any variable would have this effect. It is the purpose of the first speed-up technique introduced in this paper to eliminate some of the variables from $Cand(\alpha, h)$ that, from the point of view of the current search state, could be considered useless. To describe this elimination process with sufficient precision, let us introduce the notion of a *trace* – a partial truth-value assignment tr that records truth-value assignments made by SLS-SAT in line (2) by letting $tr(v)$ to be the new (flipped) truth-value of v each time v is selected.

Definition. Let h be a truth-value assignment. A *trace through* h is a partial truth-value assignment tr such that for every variable v, if $tr(v)$ is defined, then $tr(v) = h(v)$. By $Lit(tr)$ we shall denote the set of literals that are true under tr.

We can view DB (cf. (s3)) as a set of constraints on models for α that can be used to assess the 'quality' of this new truth-value assignment by testing the consistency status of $DB \cup Lit(tr)$. Indeed, by (s2), every model for α is a model for DB. So, if $DB \cup Lit(tr)$ is inconsistent, then some of the truth-value assignments made so far and recorded in tr have to be incorrect. We should therefore block the selection of v if such a choice leads to the inconsistency of $DB \cup Lit(tr)$. This can be achieved by 'trimming' $Cand(\alpha, h)$ using the *candidate variable trimming procedure* given in Figure 2. Let ϵ denote the trim assignment that is undefined for every variable.

procedure $Trim(Cand, h, DB, tr)$
  $C_{temp} = \emptyset$;
  for every $v \in Cand$ **do**
    $tr_{temp} = tr$; $tr_{temp}(v) = 1 - h(v)$;
    if $DB \cup Lit(tr_{temp})$ is consistent **then** $C_{temp} = C_{temp} \cup \{v\}$;
  end for
  if $C_{temp} \neq \emptyset$ **then** $Cand = C_{temp}$; **else** $tr = \epsilon$;
  return $Cand$ and tr;

Fig. 2. Candidate variable trimming procedure

The input to $Trim$ is a set of variables $Cand$, a truth-value assignment h, a set of clausal constraints DB, and a trace tr. *Trim* attempts to remove any $v \in Cand$ which would cause the inconsistency of $DB \cup Lit(tr_v)$, where tr_v is obtained from tr by assigning $1 - h(v)$ to v. If every variable in $Cand$ causes such an inconsistency, then $Cand$ is not modified (however, tr is reset to the empty assignment ϵ). For efficiency reasons, the consistency check for $DB \cup Lit(tr)$ should be done using a tractable (but incomplete) form of reasoning such as unit propagation.

The *Trim* procedure can be incorporated into the SLS-SAT algorithm by initializing DB to $\emptyset$ and tr to ϵ, and by replacing line (2) in Figure 1 with the following code:

(2a) $c_h := \bigvee\{v_h : v \in Cand(\alpha, h)\}$;
(2b) call $Trim(Cand(\alpha, h), h, DB, tr)$;
(2c) pick $v \in Cand(\alpha, h)$ using a variable selection heuristic;
(2d) $h(v) := 1 - h(v), tr(v) := h(v)$; $DB := DB \cup \{c_h\}$;

where v_h denotes the literal $\neg v$, if $h(v) = 1$ or v, otherwise. The main novelty of the resulting SLS algorithm is the restriction of the domain of the variable selection heuristic used in step (2c) from $Cand(\alpha, h)$ computed in line (1) in Figure 1 to its subset returned by the call to $Trim$ in step (2b). Let us also point out to the dynamic process of building the set DB of clauses and the trace tr in steps (2a) and (2d).

3 Literal Commitment Strategy

In a CNF SAT solver built on what is known as the Davis-Putnam-Logemann-Loveland procedure ([1]), the selection of a decision variable by a decision heuristic is followed by the assignment of truth-values to additional variables (called *implied variables*) as a result of the application of unit propagation and pure literal elimination. Such a solver is committed to these truth-vale assignments until a contradiction is detected. If one accepts the view that in a SLS procedure, a trace tr represents the truth-value assignment choices committed to by the solver, then the literals inferred from $DB \cup Lit(tr)$ can be viewed as the extent of such a commitment and the current truth-value assignment should be modified to reflect it. This strategy for modifying the current truth-value assignment beyond a single flip of the selected variable's truth-value can be implemented by replacing the assignment $h(v) = 1 - h(v)$ in line (2d) with these ones:

$$tr(v) = 1 - h(v) \text{ and for every } l \in UP(DB \cup Lit(tr)),\ h(l) = 1,$$

where $UP(DB \cup Lit(tr))$ denotes the closure of $DB \cup Lit(tr)$ under unit propagation. We shall call this version of the truth-value assignment modification the *literal commitment strategy*.

4 Experimental Evaluation

We performed series of experiments in order to determine the impact of the candidate variable trimming procedure and the literal commitment strategy on the performance of SLS SAT solvers. Although the two techniques originated from our research on non-clausal SLS methods, there is nothing intrinsically 'non-clausal' in their formulation. Hence, we evaluated the two techniques in both clausal and non-clausal settings.

As a base solver for our evaluation in the non-clausal setting we selected the SLS solver polSAT-N [12]. In polSAT-N the set $Cand(\alpha, h)$ is computed in such a way that the properties (s1) and (s2) defined in Section 1 are always satisfied. The variable selection heuristic used in polSAT-N is a non-clausal variant of the Adaptive Novelty+ [7]. By polSAT-N$^+$ we denote the solver which results from the addition of the candidate variable trimming procedure to polSAT-N. The consistency check in the $Trim$ procedure (Figure 2) is implemented by maintaining the implication graph in which the literals in the trace are used as the source nodes, and the clauses in DB are used to derive the implications – conflicts in this graph indicate the inconsistency of $DB \cup Lit(tr_{temp})$. The literal commitment strategy can be added to polSAT-N$^+$ by allowing the modification of the current truth-value assignment to be carried out using the implied literals from the graph. We denote the resulting solver by polSAT-N^{++}.

To evaluate the performance of the proposed techniques in the clausal setting, we have added the implication graph based candidate variable trimming and literal commitment strategies to the clausal SLS solver UBCSAT [13] – the resulting solver is denoted as UBCSAT^{++}. Both solvers use Adaptive Novelty+ to select variables.

The evaluation of the solvers is based on the analysis of run-length and run-time distributions (cf. [6]) on variety of instances from different benchmark classes. The non-clausal benchmarks are described in detail in [12] – we will only mention that `fs-*` and `fsf-*` are non-clausal extensions of random k-CNFs. As the non-clausal benchmarks are also available in CNF, we used them for both clausal and non-clausal evaluations. Additional CNF benchmarks come from SATLIB[1] and M.Velev's website [2]. In most cases the distributions were obtained over 250-1000 tries with the infinite cutoff – cases when this was not possible are indicated. All experiments were performed on Intel Xeon X5355, 2.66GHz, 4MB cache, 4GB RAM. The results of our experiments are presented in Table 1 – the first group of instances in the table are industrial problems, second are structured and the last are random.

Table 1. The median of the number of flips and the CPU time (sec). Cutoff is infinite – exceptions are indicated by the >cutoff and >time values. The "n/a" entry indicates that the non-CNF version of a benchmark was not available.

	non-CNF solvers						CNF solvers			
	polSAT-N		polSAT-N^{+}		polSAT-N^{++}		UBCSAT		UBCSAT^{++}	
	flips	time	flips	time	flips	time	flips	time	flips	time
`2dlx_*_005`	$> 10^7$	> 281	110375	12.42	44304	**6.10**	$> 10^8$	> 220	8442	**0.91**
`2dlx_*_017`	2521192	88.10	34539	7.54	22812	**6.03**	$> 10^8$	> 258	1779	**0.18**
`2dlx_*_049`	1873929	17.70	47666	2.60	20876	**0.96**	$> 10^8$	> 82	5794	**0.26**
`e0ddr2-*-1`	n/a		n/a		n/a		31294564	19.44	8310	**1.42**
`e0ddr2-*-4`	n/a		n/a		n/a		14726114	9.49	657	**0.14**
`parity8.easy`	212497	0.45	9215	**0.09**	5747	**0.09**	117697	**0.03**	4706	0.04
`parity8.hard`	1804016	3.67	25157	**0.19**	18120	0.27	1029140	0.26	13878	**0.10**
`logistics.d`	n/a		n/a		n/a		176000	**0.08**	2996	0.24
`qg2-08`	n/a		n/a		n/a		6221172	**22.39**	566161	31.37
`qg4-09`	n/a		n/a		n/a		120456	0.08	1503	**0.05**
`qg5-11`	n/a		n/a		n/a		8918742	11.56	1278	**0.15**
`par16-1`	n/a		n/a		n/a		113885457	31.50	781092	**4.90**
`par16-4`	n/a		n/a		n/a		396084299	110.53	884731	**5.48**
`fsf-300-*.easy`	8384	**0.11**	14392	0.55	203975	28.17	17055099	**6.46**	$> 10^7$	> 190
`fs-200-*.hard`	107737	**1.93**	246087	26.92	$> 10^6$	> 136	117011	**0.71**	5786484	274.90
`uf-250-072`	n/a		n/a		n/a		43833	**0.02**	2241390	10.45
`uf-250-093`	n/a		n/a		n/a		125165	**0.07**	2126147	9.97

The results indicate that the addition of the candidate variable trimming and the literal commitment strategies to SLS solvers significantly improves their performance on industrial SAT instances (in some cases by 2-3 orders of magnitude). Similar improvements can also be observed on many structured instances – however, in some cases the computational cost of the new techniques out-weights the improvement in the number of search steps on these instances. Finally, on random instances, the strategies in the current form seem to hurt the performance.

[1] http://www.satlib.org

[2] http://www.miroslav-velev.com/sat_benchmarks.html

5 Related Work and Final Remarks

The use of the search history, represented by a set DB of clausal constraints and a trace tr, provides a general direction for enhancing the performance of non-clausal SLS solvers with clausal techniques for SAT. It defines a class of hybrid SAT solvers whose main search heuristics still explore the input formula's original structure. However, at every step of the search for a model of an input formula α, such heuristics are refined using the search history. Since every clause in DB is an implicate of α, almost any speed-up technique developed for clausal SLS systems can be adopted to non-clausal SLS solvers. These include methods ranging from input formula simplification via the detection of dependent variables, as described in [3], to expanding DB with conflict clauses (cf. [9]) when the inconsistency of $DB \cup Lit(tr)$ is detected.

The research on SAT procedures that combine local search with reasoning (e.g. unit propagation or resolution) can be traced back to the early 1990's ([2,14]). Since then, the list of such hybrid procedures has grown significantly. The speed-up techniques presented in this paper share features with some of these hybrids, most notably with the *weak commitment search* procedure suggested in [14] and subsequently developed in [8] and [10]. The use of unit propagation to implement the literal commitment in the context of SLS was previously suggested in [4]. In [10] unit propagation was also proposed in the context of the weak commitment search. The literal commitment strategy proposed in this paper is modeled after these ideas.

Acknowledgments. We thank the anonymous referees for helpful comments.

References

1. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-Proving. Communications of the ACM 5(7), 394–397 (1962)
2. Ginsberg, M.L., McAllester, D.A.: GSAT and Dynamic Backtracking In. In: Proc. of KR 1994, pp. 226–237 (1994)
3. Grégoire, É., Ostrowski, R., Mazure, B., Saïs, L.: Automatic extraction of functional dependencies. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 122–132. Springer, Heidelberg (2005)
4. Hirsch, E.A., Kojevnikov, A.: UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. In: Annals of Mathematics and Artificial Intelligence, pp. 91–111 (2005)
5. Hoos, H.H., Stutzle, T.: Stochastic Local Search Foundations and Applications. Elsevier, Amsterdam (2005)
6. Hoos, H.H., Stutzle, T.: Local Search Algorithms for SAT: An Empirical Evaluation. J. of Automated Reasoning, 421–481 (2000)
7. Hoos, H.H.: An adaptive noise mechanism for walkSAT. In: Proc. of AAAI 2002, pp. 655–660 (2002)
8. Jussien, N., Lhomme, O.: Local Search with Constraint Propagation and Conflict-Based Heuristics. In: Proc. of AAAI 2000, pp. 169–174 (2000)
9. Lynce, I., Marques-Silva, J.P.: An Overview of Backtrack Search Satisfiability Algorithms. In: Annals of Mathematics and Artificial Intelligence, pp. 307–326 (2003)
10. Richards, E.T., Richards, B.: Nonsystematic Search and No-Good Learning. In: Journal of Automated Reasoning, pp. 483–533 (2000)

11. Selman, B., Kautz, H., Cohen, B.: Noise Strategies for Local Search. In: Proc. of AAAI 1994, pp. 337–343 (1994)
12. Stachniak, Z., Belov, A.: Speeding-up Non-Clausal Local Search for Propositional Satisfiability with Clause Learning. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 257–270. Springer, Heidelberg (2008)
13. Tompkins, D.A.D., Hoos, H.: UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 306–320. Springer, Heidelberg (2005)
14. Yokoo, M.: Weak-commitment Search for Solving Constraint Satisfaction Problems. In: Proc. of AAAI 1994, pp. 313–318 (1994)

A Theoretical Analysis of Search in GSAT

Evgeny S. Skvortsov

School of Computing Science
Simon Fraser Univerity
evgenys@sfu.ca

Abstract. This paper is devoted to a rigorous analysis of the GSAT algorithm in the typical case for the random planted 3-SAT distribution. GSAT was the first widely appreciated practical heuristic developed for SAT that was based on the local search principles. We show that for any constant $\kappa > 0$ GSAT, with high probability, solves random planted 3-SAT problems of density $\rho = \kappa \ln n$. This performance is substantially better than the performance of the pure Iterative Improvement algorithm that has a phase transition at $\rho = \frac{7}{6} \ln n$ and fails for problems of smaller density.

1 Introduction

The GSAT algorithm was proposed in the early 90s by Selman, Levesque and Mitchell [13]. This algorithm starts its work with a random assignment and at every step flips one of the variables that give the maximum increase in the number of satisfied clauses (the maximum possible increase can happen to be zero or even negative). If a satisfying assignment is not found for a certain number of steps then the algorithm restarts. It was demonstrated in the same paper that GSAT outperformed state-of-the-art systematic search algorithms of the time. Extensive empirical analysis of GSAT was carried out by Gent et al. [5].

Later many algorithms were built on the basis of GSAT. In particular Gent and Walsh have experimentally demonstrated [7,6] that greediness is not very important for success of GSAT. The algorithm they designed, CSAT, starts with a random assignment and then at every step flips a variable chosen uniformly at random among ones that give any increase in the number of satisfied clauses. If no such variable exists then a variable is chosen among those that do not change the number of satisfied clauses (that is a *plateau move* is performed). Similarly to GSAT if a satisfying assignment is not found after a certain number of steps CSAT restarts.

Worst case efficiency of GSAT applied to SAT when time given to solve the problem is polynomial is not better than performance of Iterative Improvement [10] (the basic Local Search algorithm which can be viewed as CSAT with neither plateau moves nor restarts). It guarantees only $\frac{k}{k+1}$-th fraction of clauses to become satisfied [8,12], where k is the length of the shortest clause. Worst case efficiency of the GSAT and CSAT algorithms for formulas of bounded clause to variable ratio in the settings of exponentially long run was studied by Hirsch.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 265–275, 2009.

In particular exponential upperbound of the form $2^{cn}, c < 1$ was shown for time of the execution of CSAT as a Monte-Carlo randomized algorithm. This upperbound was obtained for its execution without use of plateau moves.

Typical case efficiency of SAT algorithms is often studied for two distributions: *random 3-SAT of fixed density* and *random planted 3-SAT of fixed density*. Random 3-SAT problem of density $\rho = \rho(n)$ is sampled uniformly at random from the set of all 3-CNFs with n variables and $\rho(n) \cdot n$ clauses. To sample a random planted 3-SAT one first fixes a boolean assignment of values to variables and the 3-CNF is sampled uniformly at random from the set of all 3-CNFs that are satisfied by this assignment.

A positive result about efficiency of GSAT in the typical case for a random planted 3-SAT was proven by Koutsoupias and Papadimitriou [11]. It was shown that for linear density $\rho = \kappa n$ GSAT succeeds with high probability. As in the proof of the upper bound of the time complexity in [9] plateau and downward moves were not used in the analysis. Also authors formulated conjecture that their upper bound is not tight and that GSAT can solve random planted 3-SAT for a large constant density.

Later Gent [4] has adapted techniques of Koutsoupias and Papadimitriou [11] to show that even for *Stupid Algorithm* (i. e. an algorithm that assigns 1 or 0 to a variable depending on whether this variable occurs more often in positive or negative literals) there exists a large constant κ_0 such that this algorithm can solve random planted 3-SAT for density $\kappa \ln n$, for any $\kappa > \kappa_0$.

Recently, in a joint work with Bulatov, we have shown [1] that the efficiency of Iterative Improvement has a phase transition at density $\rho_0 = \frac{7}{6} \ln n$. That is for any positive constant ε if we have $\rho > \rho_0 \cdot (1+\varepsilon)$ then Iterative Improvement, with high probability, finds a solution for random planted 3-SAT of density ρ. For $\rho < \rho_0 \cdot (1-\varepsilon)$ Iterative Improvement with high probability fails. It is not hard to estimate that the constant κ for which the *Stupid Algorithm* works is orders larger that $\frac{7}{6}$.

In this paper we show that for any constant $\kappa > 0$ GSAT and CSAT can solve random planted 3-SAT of density $\rho = \kappa \ln n$ without restarts. The result rigorously proves that using plateau moves substantially increases efficiency of Iterative Improvement when applied to random 3-SAT with a planted solution.

Note that there are algorithms that were proven to solve random planted 3-SAT for densities smaller than logarithmic. A spectral heuristic algorithm was developed by Flaxman [3] with the specific purpose of solving random planted 3-SAT and its modifications of high constant density. So there exists a constant ρ_0 such that spectral heuristic solves random planted 3-SAT of any constant density ρ, such that $\rho > \rho_0$. Later Feige and Vilenchik [2] developed a more intuitive local search based algorithm with similar performance on random planted 3-SAT that was also demonstrated to be robust against a certain type of non-randomness of the problem. Our goal in this paper is not to develop a better algorithm for random planted 3-SAT, but to move closer to a theoretical understanding of specific reasons that make algorithms used in practice to be as efficient as they are.

The rest of the paper is organized as follows. In Section 2 we give necessary definitions, in Section 3 we prove the main result, we conclude and discuss directions of future work in Section 4.

2 Definitions

A literal is a boolean variable or a negation of a boolean variable. A disjunction of k literals corresponding to distinct variables is called a *k-clause*. A 3-CNF is a conjunction of *3-clauses*. As we consider only 3-CNFs, we will always call them just clauses. We assume that variables in the 3-CNF are $x_1, \ldots, x_n$ and we consider any boolean vector $\boldsymbol{v}$ of dimension n as an assignment $\{x_i = v_i\}_{i=1\ldots n}$.

A *random 3-CNF* distribution $\Phi(n, \rho(n)n)$ is a uniform probability measure and its state space is a set of all 3-CNFs with n variables and $\lfloor \rho(n)n \rfloor$ clauses. In other words a random 3-CNF $\phi \in \Phi(n, \rho(n)n)$ is generated by choosing independently uniformly at random $\lfloor \rho n \rfloor$ out of $8\binom{n}{3}$ all possible clauses.

A *random 3-CNF with a planted solution* $\boldsymbol{v}$ distribution is obtained from a random 3-CNF by restricting its state space to a set of 3-CNFs that are satisfied by $\boldsymbol{v}$. We can sample random planted 3-CNF $\phi \in \Phi^{\mathtt{plant}}(n, \rho(n)n, \boldsymbol{v})$ by choosing $\lfloor \rho(n)n \rfloor$ out of all $7\binom{n}{3}$ clauses that are satisfied by $\boldsymbol{v}$. It is convenient to assume that $\boldsymbol{v} = (1, \ldots, 1)$ and shorten the notation and the name of this distribution to $\Phi^{\mathtt{plant}}(n, \rho n)$ and *random planted 3-CNF* respectively.

We call a clause c a $(+,-,-)$ clause if one of its literals is positive and other two are negative. That is, c is a horn clause.

For $\phi \in \Phi^{\mathtt{plant}}(n, \rho n)$ we say that a statement $\mathcal{E}(\phi)$ is true *with high probability* if probability of the event $\mathcal{E}(\phi)$ tends to 1 for $\phi \in \Phi^{\mathtt{plant}}(n, \rho n), n \to \infty$. We shall also use standard acronym "whp".

For arbitrary functions $f(n), g(n)$ we denote equality $f(n) = g(n) + o(n)$ by $f(n) \approx g(n)$ and we write $f(n) \lesssim g(n)$ if inequality $f(n) \leq g(n)$ holds for large enough n.

As it was discussed in [9,1] there are two kinds of local maxima of a random 3-CNF with a planted solution. Some maxima have almost all variables equal to zero and others have almost all variables equal to one. Maxima that have almost all variables equal to zero satisfy extremely few clauses. GSAT starts with a random assignment that has around $\frac{7}{8}$th fraction of clauses satisfied and never reaches this kind of maxima. So it will be convenient for us to say "assignment $\boldsymbol{v}$ is a local maximum" for "assignment $\boldsymbol{v}$ is a local maximum of the number of satisfied clauses that has more than $\frac{1}{10}$th fraction of variables assigned to one".

Let $\boldsymbol{v}'_i$ be a vector obtained from $\boldsymbol{v}$ by flipping i-th coordinate. Statements "c is a clause in ϕ" and "clause c is satisfied by $\boldsymbol{v}$" we denote by $c \in \phi$ and $c(\boldsymbol{v})$ respectively. The GSAT algorithm is presented at Fig. 1.

3 Main Result

Now the main result of this paper is easy to formulate.

INPUT: A 3-CNF ϕ, integers MAXTRIES, MAXFLIPS
OUTPUT: *fail* or an assignment $\boldsymbol{v}$ that satisfies ϕ
METHOD:
do MAXTRIES times
pick $\boldsymbol{v}$ uniformly at random
do MAXFLIPS times
if $\boldsymbol{v}$ satisfies ϕ **then return** $\boldsymbol{v}$
pick a variable x_i such that $|\{c \in \phi \mid c(\boldsymbol{v}'_i)\}|$ is maximal uniformly at random
let $\boldsymbol{v} = \boldsymbol{v}'_i$
return *fail*

Fig. 1. The GSAT algorithm

Theorem 1. *For any* $\kappa > 0, \rho = \kappa \ln n$ *the GSAT algorithm with settings* MAXFLIPS$= n^{60/\kappa+3}$, MAXTRIES$= 1$ *finds a solution for* $\phi \in \Phi^{\mathtt{plant}}(n, \rho n)$ *whp.*

After $O($MAXFLIPS$\times$MAXTRIES$)$ steps that did not lead to a solution of the problem GSAT fails so Theorem 1 means that GSAT finds solution in polynomial number of steps.

It is sufficient to prove the result for $\kappa < 2$, since by Teorem 1 of [1] for $\kappa > \frac{7}{6}$ GSAT will find solution without even switching to plateau moves stage.

Lemmas 1-3 describe relations between variables that are assigned to 0 in a local maximum and variables that occur in few $(+,-,-)$ clauses. These lemmas lead to a proof of Lemma 4 that will be the key instrument to prove Theorem 1. This lemma is formulated in terms of a graph of co-occurrences that we define later. In terms of the original formula ϕ Lemma 4 states that when a local maximum of the number of satisfied clauses is reached we have the following picture. Clauses containing variables that are still assigned incorrectly fall apart into several sub-formulas $\phi_1, \ldots, \phi_t$. Formulas $\phi_1, \ldots, \phi_t$ are pairwise disjoint, that is no ϕ_i, ϕ_j contain a common clause. Moreover these sub-formulas are disjoint with respect to variables, that is no ϕ_i, ϕ_j refer to a common variable. The lemma also states that any such sub-formula ϕ_i that contains an unsatisfied clause contains an incorrectly assigned variable that can be flipped by GSAT. In the proof of the Theorem 1 we apply the Lemma 4 and observe that the landscape of the value function is with high probability such that from any proper local maximum there is a finite path along plateau that leads to a higher ground. The path is finite meaning that its length can be bounded by some constant ℓ that does not depend on n. So with constant probability among next ℓn^{ℓ} steps there will be ℓ that will be made along such a path and a better assignment will be reached.

We will discuss intuition behind Lemmas 1-3 directly before formulating them. Next we define a *graph* G^{ϕ} *of co-occurrences* of variables in clauses of ϕ and several related notions.

Let ϕ be a CNF and E be a set of pairs of variables (x_i, x_j) such that x_i and x_j occur in the same clause. We denote a graph $(\{x_1, \ldots, x_n\}, E)$ by G^{ϕ}. Since formula ϕ will always be clear from context we will omit upper index ϕ and introduce all further notation without it.

For a graph $G = (V, E)$ and a subset of its vertices X we denote by $G|_X$ a subgraph induced by X, i.e. graph $(X, E \cap X^2)$. Let $l \in \mathbb{N}$ and E^l be a set of pairs of vertices that are connected by paths of length at most l in G. We denote graph (V, E^l) by G^l. We denote by $N^{G,d}(X)$ a d-th neighborhood of X in G, i.e.

$$N^{G,d}(X) = \{y \mid \text{there exists } x \in X \text{ such that } (x, y) \in E^d\}.$$

Observation 1. *For any $\rho \leq 2 \ln n$ whp no two variables occur together in more than 2 clauses of $\phi \in \Phi^{\mathtt{plant}}(n, \rho n)$. That is, whp any two vertices of G are connected by at most 2 edges.*

Proof. Indeed if we fix two variables and three clauses then probability of the variables to occur together in these three clauses is $O(n^{-6})$. There are $O(n^2)$ pairs of variables and $O(m^3)$ triples of clauses so applying union bound we conclude that probability that there exist such pair and triple is less than $O(n^{-4}m^3)$ which tends to zero for $m \leq 2n \ln n$. □

We say that a variable is Δ-isolated in ϕ if it occurs positively in less than Δ clauses of type $(+, -, -)$. We denote a set of all Δ-isolated variables by I_Δ.

Intuitively, our interest in Δ-isolated clauses comes from the fact that in the case of logarithmic density of the formula extremely few clauses with two or three positive literals are unsatisfied in a local maximum. This happens because almost all variables are assigned correctly and probability that a clause has two incorrect variables is very small. Thus many unsatisfied clauses in a local maximum are $(+, -, -)$ and variables that are assigned wrong tend to be Δ-isolated variables. In the first Lemma of this paper we show that Δ-isolated variables do not "flock together" so with high probability you do not find many of them close to each other.

Lemma 1. *For any $\kappa \in \mathbb{R}$ there is a constant $l \in \mathbb{N}$ such that for any constants $\Delta \in \mathbb{N}, d \in \mathbb{N}$ and $\rho = \kappa \ln n, \phi \in \Phi^{\mathtt{plant}}(n, \rho n)$ whp any connected component of $G^d|_{I_\Delta}$ contains less than l variables.*

Proof. Fix an arbitrary constant r. Let M be a set of r variables. Obviously the probability of the event

$$\text{all variables in } M \text{ are in } I_\Delta$$

is less than the probability of the event

$$\#(\text{positive occurrences of variables from } M \text{ in } (+, -, -)\text{-clauses}) \leq |M| \times \Delta.$$

The latter probability can be bounded above by

$$\sum_{k=0}^{r\Delta} \binom{\rho n}{k} \left(\frac{r}{n}\right)^k \left(1 - \frac{3r}{7n}\right)^{\rho n - k} \leq \tag{1}$$

$$r\Delta \binom{\rho n}{r\Delta} \left(\frac{r}{n}\right)^{r\Delta} \left(1 - \frac{3r}{7n}\right)^{\rho n - r\Delta} \lesssim \tag{2}$$

$$r\Delta \rho^{r\Delta} \Delta^{-r\Delta} e^{-3/7\rho r} / ((r\Delta)!) \lesssim e^{-3\rho r/14}. \tag{3}$$

The number of connected sets of variables in G^d can be bounded above by the number of subgraphs of G^d isomorphic to trees. Since whp maximum degree of a variable is bounded above by $\ln^2 n$ the number of subgraphs of size r isomorphic to trees can be bounded above by $n(r\ln^{2d} n)^{r-1} \lesssim n^2$ whp.

Now we apply union bound to the probability of an event that there exists a connected set M of vertices of G^d_ϕ of size r such that all variables in M are in I_Δ getting the upper bound

$$n^2 e^{-3\kappa r \ln n/14} = e^{\ln n(2-3\kappa r/14)}.$$

Therefore if we set $l = 10/\kappa$ then for any constant $r, r > l$ we have the probability tending to 0. □

For an assignment $\boldsymbol{v}$ we denote by $W_{\boldsymbol{v}}$ a set of all variables assigned by $\boldsymbol{v}$ incorrectly. I. e.

$$W_{\boldsymbol{v}} = \{x_i \mid v_i = 0\}.$$

In Lemma 2 we show that in a local maximum any connected component of a graph of co-occurenses of variables assigned incorrectly needs a constant fraction of its members to belong to I_Δ.

Lemma 2. *For any $\kappa \in \mathbb{R}$ and odd $\Delta \geq 11$ for $\rho = \kappa \ln n, \phi \in \Phi^{\mathtt{plant}}(n, \rho n)$ whp for any local maximum $\boldsymbol{v}$ any connected component C of $G|_{W_{\boldsymbol{v}}}$ contains at least $\frac{|C|(\Delta-9)}{\Delta+1}$ variables from I_Δ.*

Proof. By Lemma 3 from [1] there exists α such that $0 < \alpha < 1$ and any local maximum contains less than n^α zeros. Consider an arbitrary local minimum $\boldsymbol{v}$ and a connected component C of $G|_{W_{\boldsymbol{v}}}$.

Let x_i be such that $x_i \in C \setminus I_\Delta$. Variable x_i occurs positively in at least Δ clauses of type $(+,-,-)$. If clause c is of type $(+,-,-)$, variable x_i occurs positively in c and it is the only variable in c that is assigned to 0 then c is not satisfied and will become satisfied if x_i is flipped. But $\boldsymbol{v}$ is a local maximum and flipping x_i should not increase number of satisfied clauses. If x_i has no neighbors assigned to 0 by $\boldsymbol{v}$ then after flipping x_i number of satisfied clauses will increase by at least Δ. A neighbor x_j of x_i may decrease this advantage by making one of $(+,-,-)$ clauses that refer to x_i satisfied or by making some other clause where x_i occurs negatively unsatisfied. But each neighbor x_j assigned to 0 can not decrease the advantage of flipping of x_i by more than the number of co-occurences of x_i and x_j in clauses of ϕ. By Observation 1 whp no two variables occur together in more than 2 clauses. So if x_i has t neighbors assigned to 0 then advantage of flipping x_i will be at least $\Delta - 2t$. And since the advantage must be non positive and is an integer number we have $t \geq (\Delta+1)/2$.

Therefore x_i must have at least $(\Delta+1)/2$ neighbors in G that are assigned to zero by $\boldsymbol{v}$. Obviously all these variables are in C, so degree of x_i in $G|_{W_{\boldsymbol{v}}}$ is at least $(\Delta+1)/2$. We can bound average degree of C from below by $(|C \setminus I_\Delta| \cdot (\Delta+1)/2)/|C|$. Since $|C| < n^\alpha$ by Lemma 1(1) from [1] it follows that average degree of C is less than 5. Thus we have

$$\left(|C \setminus I_\Delta| \cdot (\Delta+1)/2\right)/|C| < 5$$

and consequently

$$|C \setminus I_\Delta| < \frac{10|C|}{\Delta+1}, \tag{4}$$

$$|C \cap I_\Delta| > \frac{|C|(\Delta-9)}{\Delta+1}. \tag{5}$$

Since inequality (5) was shown for an arbitrary connected component of $G|_{W_{\boldsymbol{v}}}$ the Lemma is proven. □

We say that a variable x_i is *potentially wrong* if there is an assignment $\boldsymbol{v}$ such that it is a local minimum and $v_i = 0$. A set of all potentially wrong variables is denoted by W, that is

$$W = \bigcup_{\boldsymbol{v} \text{ is a local maximum}} W_{\boldsymbol{v}}.$$

In the following lemma we show that for large enough Δ the set of Δ-isolated variables is dense in the set of potentially wrong variables. Namely that any potentially wrong variable must have at least one Δ-isolated variable in a finite distance.

Lemma 3. *For any $\kappa \in \mathbb{R}$ there is a constant $r \in \mathbb{N}$ such that for $\rho = \kappa \ln n$, $\phi \in \Phi^{\texttt{plant}}(n, \rho n)$, and odd $\Delta, \Delta \geq 11$, whp any $x_i \in W$ has a Δ-isolated variable at distance less than r.*

Proof. Let l be the number corresponding to κ by Lemma 1, let $r = 7l$ and let

$$M = \{x | x \text{ is connected in } G^r \text{ to some } y \in I_\Delta\} = N^{G,r}(I_\Delta).$$

Consider an arbitrary local maximum $\boldsymbol{v}$. To prove the lemma we must show that

$$W_{\boldsymbol{v}} \subseteq M.$$

For the sake of contradiction let us assume that

$$\text{there exists } x_j \in W_{\boldsymbol{v}} \setminus M. \tag{6}$$

Let C be a connected component of $G|_{W_{\boldsymbol{v}}}$ containing x_j. By Lemma 2 set C contains at least $\frac{|C|(\Delta-9)}{\Delta+1}$ variables from I_Δ. For $\Delta \geq 11$ we have

$$|C \cap I_\Delta| \geq 1/6|C| \tag{7}$$

and consequently nonempty C must contain at least 1 variable form I_Δ. Now we take arbitrary $x_k \in I_\Delta \cap C$ and consider a connected component C' of $G^{2r}|_{I_\Delta}$ that contains x_k. Note that Lemma 1 implies

$$|N^{G,r}(C') \cap I_\Delta \cap C| \leq |N^{G,r}(C') \cap I_\Delta| \leq l.$$

On the other hand since $x_j \notin M \supseteq N^{G,r}(C')$ and x_j is connected to an element of C' we have

$$|N^{G,r}(C') \cap C| \geq r = 7l.$$

By definitions of $N^{G,d}$ and G^d any distinct connected components C_1 and C_2 of $G^{2r}|_{I_\Delta}$ satisfy

$$N^{G,r}(C_1) \cap N^{G,r}(C_2) = \varnothing.$$

Therefore if $s > 0$ is the number of connected components of $G^{2r}|_{I_\Delta}$ that intersect with C we have bounds

$$|I_\Delta \cap C| \leq sl \tag{8}$$

and

$$|C| \geq 7sl. \tag{9}$$

Conjunction of (8) and (9) contradicts (7), therefore assumption (6) was false and the lemma is proven. □

By now we are in a position to prove the lemma from which the main result will rather easily follow. As it was discussed in the beginning of the section intuitive meaning of Lemma 4 is that in a local maximum we have variables assigned incorrectly split into several finite connected components. Moreover each component that contains a variable occurring in an unsatisfied clause contains also a variable that can be flipped by GSAT.

Lemma 4. *For any $\kappa \in \mathbb{R}$ there is $s \in \mathbb{N}$ such that for $\rho = \kappa \ln n, \phi \in \Phi^{\mathtt{plant}}(n, \rho n)$ whp for any local maximum $\boldsymbol{v}$ and any connected component C of $G|_{W_{\boldsymbol{v}}}$ the following is true:*

- *C contains less than s elements,*
- *if there is an unsatisfied clause containing a variable from C then there is a variable $x_j \in C$ such that the number of unsatisfied clauses where x_j occurs equals to the number of clauses that are satisfied only by x_j.*

Proof. We fix some local maximum $\boldsymbol{v}$ and a connected component C of $G|_{W_{\boldsymbol{v}}}$.

By Lemma 3 for $\Delta = 11$ there is a constant r such that whp for every variable $x_i \in C$ there is a variable $x_j \in I_\Delta$ such that distance between x_i and x_j is less than r. Let us denote such x_j by $x_i \uparrow$. It is easy to see that set $\{x_i \uparrow | x_i \in C\} \cup (C \cap I_\Delta)$ is connected in $G^{2r+1}|_{I_\Delta}$ and by Lemma 1 its size can not be greater than some constant l. Thus $|C \cap I_\Delta| < l$ and by Lemma 2 for $\Delta = 11$ we have $|C| < 6l$. So we set $s = 6l$ and have the first statement of the Lemma proven. Note that in the proof of Lemma 2 we set $l = 10/\kappa$ so here we have $s = 60/\kappa$.

To prove the second statement of the Lemma we consider an arbitrary variable x_j in C and a clause c where x_j occurs. We can make the following

Observation 2. *For c to be satisfied only by x_j in $\boldsymbol{v}$ the following two conditions are necessary: (a) x_j occurs in c negatively, (b) there is a variable x_i that occurs in c positively and such that $v_i = 0$.*

Consider a directed graph

$$\mathcal{S}_C = (C, \{(x_i, x_j) | \text{there is a clause } c \in \phi, \text{ containing literals } x_i \text{ and } \neg x_j\}).$$

Assume for the sake of contradiction that for any variable $x_j \in C$ the number of clauses that are satisfied only by x_j is strictly greater than the number of unsatisfied clauses where x_j occurs. Then for each $x_j \in C$ there is at least one clause that is satisfied only by x_j. Which by Observation 2 means that the in-degree of every vertex in $\mathcal{S}_C$ is at least 1. Set C contains a variable x_k that occurs at some unsatisfied clause c so there must be at least two clauses that are satisfied only by x_k. Thus the in-degree of x_k in $\mathcal{S}_C$ is at least 2 and $\mathcal{S}_C$ contains at least $|C|+1$ edges. If variables are connected in $\mathcal{S}_C$ they are connected in $G|_C$ and we have that $G|_C$ contains at least $|C|+1$ edges.

We finish the proof by showing that for any constants h and q such that $h > q$ whp there is no set of variables C such that $|C| = q$ and the graph $G|_C$ contains h edges. Indeed there are $\binom{n}{q}$ sets of variables of size q and $\binom{m}{h}$ sets of clauses of size h. For a given set of clauses of size h the probability to have $2h$ positions to be occupied by variables from a given set $C, |C| = q$ can be bounded above by $(3h)^{2h} n^{-2h}$. Applying union bound we have that the probability under consideration is less than $(3h)^{2h} m^h n^q n^{-2h}$ which tends to 0 for any fixed κ if $h > q$.□

We are now in a position to prove the main result.

Proof. (of Theorem 1) By Lemma 4 once GSAT reaches a local maximum $\boldsymbol{v}$ set $W_{\boldsymbol{v}}$ falls apart into several connected components of size at most s. If there are no more unsatisfied clauses left then the problem is solved and GSAT returns a satisfying assignment. Otherwise let us consider a connected component C of $G|_{W_{\boldsymbol{v}}}$ that contains a variable x_i that occurs in an unsatisfied clause.

Now we show that with probability at least n^{-s} after s steps GSAT will be at some assignment $\boldsymbol{u}$ that satisfies more clauses than $\boldsymbol{v}$. By Lemma 4 there is a variable $x_{i_1} \in C$ such that the number of clauses that are satisfied only by x_{i_1} equals the number of clauses that contain x_{i_1} and are not satisfied. Thus with probability $1/n$ variable x_{i_1} will be the next variable that is flipped by GSAT. If it happens then for an assignment $\boldsymbol{v}'$ obtained at the next step there are two possibilities: 1) $\boldsymbol{v}'$ is not a local maximum or 2) $\boldsymbol{v}'$ is still a local maximum. In case 1) $GSAT$ will increase the number of satisfied clauses at the next step. In case 2) let C_1 be a subset of $C \setminus \{x_{i_1}\}$ that is a connected component of $G|_{W_{\boldsymbol{v}}}$ and contains variable that occurs in an unsatisfied clause. We have $|C_1| \leq |C| - 1$ and with probability $1/n$ a variable from C_1 will be flipped by GSAT at the next step, which will either lead to an increase of the number of satisfied clauses or to a new set $C_2, |C_2| \leq |C| - 2$, etc. Size of C is at most s so with probability greater than n^{-s} after s steps number of satisfied clauses will be increased.

Therefore if GSAT is at a local maximum then in sn^{s+1} steps it will increase number of satisfied clauses with probability at least $1 - (1 - n^{-s})^{n^{s+1}} \approx 1 - e^{-n}$. So once the local maximum is reached for the first time the problem will be solved after sn^{s+2} steps with probability at least $1 - ne^{-n}$. □

4 Conclusion and Future Work

Note that in no proof we used greediness of GSAT and all the reasoning would go in the very same way for CSAT. Therefore we can formulate

Corollary 1. *For any* $\kappa > 0, \rho = \kappa \ln n$ *the CSAT algorithm with settings* MAXFLIPS$= n^{60/\kappa+3}$, MAXTRIES$= 1$ *finds a solution for* $\phi \in \Phi^{\mathtt{plant}}(n, \rho n)$ *whp.*

Comparing Corollary 1 with Theorem 1 in [1] and noting that CSAT is the Iterative Improvement (basic Local Search) enhanced with plateau moves and restarts we can conclude that adding plateau moves to Iterative Improvement increases its power substantially. The intuitive essence of this conclusion is by no means novel but now we have it rigorously proven within the context of random planted 3-SAT.

We believe that the analysis of the landscape of the value function of random planted 3-SAT carried out in this paper gives more general intuitive understanding of the process of the execution of the local search algorithms. Possible directions of the future work include

- exploration of the efficiency of GSAT for random planted 3-SAT of smaller densities,
- the analysis of GSAT for the uniform random 3-SAT problem.

It was experimentally shown [13] that GSAT works well for constant densities for random 3-SAT and this fact must have a theoretical explanation.

Acknowledgement. The author is grateful to his supervisors Andrei Bulatov and David Mitchell for multiple fruitful discussions of the topic.

References

1. Bulatov, A.A., Skvortsov, E.S.: Phase transition for local search on planted SAT. CoRR, abs/0811.2546 (2008)
2. Feige, U., Vilenchik, D.: A local search algorithm for 3SAT. Technical report MCS04-07 of the Weizmann Institute (2004)
3. Flaxman, A.: A spectral technique for random satisfiable 3CNF formulas. In: SODA, pp. 357–363 (2003)
4. Gent, I.: On the stupid algorithm for Satisfiability. Ian. Gent. APES Report APES-03-1998. APES Research Group (1998)
5. Gent, I.P., Bridge, S., Walsh, T.: An empirical analysis of search in GSAT. Journal of Artificial Intelligence Research 1, 47–59 (1993)
6. Gent, I.P., Walsh, T.: The enigma of SAT hill-climbing procedures. Technical report, Department of AI, University of Edinburgh (1992)
7. Gent, I.P., Walsh, T.: Towards an understanding of hill-climbing procedures for SAT. In: AAAI, pp. 28–33 (1993)
8. Hansen, P., Jaumard, B.: Algorithms for the maximum Satisfiability problem. Computing 44, 279–303 (1990)
9. Hirsch, E.A.: SAT local search algorithms: Worst-case study. J. Autom. Reason 24(1-2), 127–143 (2000)

10. Hoos, H., Sttzle, T.: Stochastic Local Search: Foundations & Applications. Morgan Kaufmann Publishers Inc., San Francisco (2004)
11. Koutsoupias, E., Papadimitriou, C.: On the greedy algorithm for Satisfiability. Inf. Process. Lett. 43(1), 53–55 (1992)
12. Mastrolilli, M., Gambardella, L.M.: Maximum satisfiability: How good are tabu search and plateau moves in the worst-case? European Journal of Operational Research 166(1), 63–76 (2005)
13. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: AAAI, pp. 440–446 (1992)

The Parameterized Complexity of k-Flip Local Search for SAT and MAX SAT

Stefan Szeider

Department of Computer Science, Durham University,
Durham DH1 3LE, England, United Kingdom
stefan.szeider@durham.ac.uk

Abstract. SAT and MAX SAT are among the most prominent problems for which local search algorithms have been successfully applied. A fundamental task for such an algorithm is to increase the number of clauses satisfied by a given truth assignment by flipping the truth values of at most k variables (k-flip local search). For a total number of n variables the size of the search space is of order n^k and grows quickly in k; hence most practical algorithms use 1-flip local search only. In this paper we investigate the worst-case complexity of k-flip local search, considering k as a parameter: is it possible to search significantly faster than the trivial n^k bound? In addition to the unbounded case we consider instances with a bounded number of literals per clause or where each variable occurs in a bounded number of clauses. We also consider the related problem that asks whether we can satisfy *all* clauses by flipping the truth values of at most k variables.

1 Introduction

Local search (LS) is one of the most fundamental algorithmic concepts and has been successfully applied to a wide range of hard combinatorial optimization problems, most prominently to Maximum Satisfiability (MAX SAT) and the Travelling Salesperson Problem (TSP). The basic idea is to move—as long as possible—from a candidate solution to a "better" neighboring candidate solution. For MAX SAT the candidate solutions are truth assignments; two truth assignments are *k-flip neighbors* if they differ in the values of at most k variables; a truth assignment is better than the other if it satisfies more clauses. Numerous sophisticated variants of the basic LS algorithm for MAX SAT have been suggested in the literature; for example LS algorithms that, if stuck at a local maximum, heuristically move to a non-improving solution. An in-depth coverage LS algorithms can be found in Hoos and Stützle's book [6].

The number of k-flip neighbors of a truth assignment on n variables is of order n^k, a size that grows rapidly in k. It is therefore not surprising that most practical algorithms consider 1-flip neighborhoods only; already 2- or 3-flip neighborhoods are too large for a brute forth search, as typical instances have tens or hundreds of thousands of variables.

In this paper we study the question of whether the k-flip neighborhood can be exhaustively searched in a more efficient way. In particular, we investigate whether the search can be carried out within a worst-case time bound that is polynomial for fixed k where the order of the polynomial is independent of k (in contrast to the n^k time bound as required by brute forth search). Problems that admit an algorithmic solution of this

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 276–283, 2009.

type are called *fixed-parameter tractable* (FPT). Whether or not a problem is fixed-parameter tractable is studied in the theoretical framework of Parameterized Complexity [2,4,12,14]; we provide some basic definitions and concepts in Section 2.2. We study the parameterized complexity of LS for MAX SAT in general and for special cases where clause-size or the number of occurrences of variables are bounded. Furthermore we study the parameterized complexity of a related problem where we ask whether a k-flip neighbor of the current truth assignment satisfies all clauses (i.e., if there is a full solution of distance at most k from the current one). More specifically, we consider the following two problems and special cases thereof with bounds on clause-size and the occurrence of variables.

k-FLIP MAX SAT
Instance: A CNF formula F and a truth assignment $\tau : \mathrm{var}(F) \to \{0,1\}$.
Question: Is there a k-flip neighbor τ' of τ that satisfies more clauses of F than τ?

k-FLIP SAT
Instance: A CNF formula F and a truth assignment $\tau : \mathrm{var}(F) \to \{0,1\}$.
Question: Is there a k-flip neighbor τ' of τ that satisfies all clauses of F?

The following table summarizes our results ("FPT" indicates fixed-parameter tractability, "W[1]-hard" and "W[2]-hard" indicate that the considered problem is most likely not fixed-parameter tractable, see Section 2.2).

size of clauses	occurrence of variables	k-FLIP MAX SAT	k-FLIP SAT
unbounded	unbounded	W[1]-hard	W[2]-hard
unbounded	bounded	W[1]-hard	W[1]-hard
bounded	unbounded	W[1]-hard	FPT
bounded	bounded	FPT	FPT

Related Work. k-flip LS plays an important role in various theoretical investigations, for example in Dantsin et al.'s work on worst-case upper bounds for the running time of 3-SAT algorithms [1]. The *expected* running time for searching 2- and 3-flip neighborhoods on random instances has been investigated by Yagiura and Ibaraki [15]. Johnson, Papadimitriou, and Yannakakis [8] introduced the class PLS of LS problems for which local optimality can be verified in polynomial time, and showed that there are complete problems for this class.

The study of the parameterized complexity of LS was initiated by Fellows [3]. To date only a handful results are known: Khuller, Bhatia, and Pless [9] investigated the problem of finding a feedback edge set in a graph that is incident to as few vertices as possible. They showed that the search for a better solution which can be obtained by replacing at most k edges of the feedback edge set is fixed-parameter tractable. Marx [11] studied the parameterized complexity of LS for TSP. He established that finding a better tour by replacing at most k arcs of a given tour is W[1]-hard, even when the distance matrix is symmetric and satisfies the triangle inequality. The parameterized complexity of the important special case where the cities are points in the Euclidean

plane remains open. Krokhin and Marx [10] investigated the parameterized complexity of LS for finding a satisfying truth assignment for a Boolean constraint satisfaction instance that sets as few variables as possible to 0. They established a dichotomy theorem (similar to Schaefer's Theorem) that exactly characterizes which classes of Boolean relations allow fixed-parameter tractability and which do not.

2 Preliminaries

2.1 CNF Formulas and Truth Assignments

We consider propositional formulas in conjunctive normal form, *CNF formulas*, given as sets of clauses. A *clause* is a set of literals, a *literal* is a propositional variable x (a positive literal) or a negated variable $\neg x$ (a negative literal). A CNF formula F is a q-CNF formula if each clause of F contains at most q literals. We say that a variable x *occurs* in a clause C if $x \in C$ or $\neg x \in C$. The *variable occurrence* of a CNF formula F is bounded by an integer p if each variable x of F occurs in at most p clauses of F. We write $\mathrm{var}(F)$ for the set of variables that occur in F. A *truth assignment* is a mapping $\tau : X \to \{0,1\}$ defined on a set X of variables. A truth assignment τ *satisfies* a clause C if $\tau(x) = 1$ for some $x \in C$ or $\tau(x) = 0$ for some $\neg x \in C$; τ satisfies a CNF formula F if it satisfies all clauses of F. Let $\tau : \mathrm{var}(F) \to \{0,1\}$ and $\tau' : \mathrm{var}(F) \to \{0,1\}$ be truth assignments. We define $\mathrm{dist}(\tau,\tau') = |\{\, x \in \mathrm{var}(F) : \tau(x) \neq \tau(x') \,\}|$ and $\mathrm{sat}(\tau, F) = |\{\, C \in F : \tau \text{ satisfies } C \,\}|$. If $\mathrm{dist}(\tau,\tau') \leq k$ then we say that τ and τ' are *k-flip neighbors*.

2.2 Parameterized Complexity

An instance of a parameterized problem is a pair (I,k) where I is the *main part* and k is the *parameter*; the latter is usually a non-negative integer. A parameterized problem is *fixed-parameter tractable* if there exist a computable function f and a constant c such that instances (I,k) can be solved in time $O(f(k)\|I\|^c)$ where $\|I\|$ denotes the size of I. FPT is the class of all fixed-parameter tractable decision problems.

A *parameterized reduction* is a many-one reduction where the parameter for one problem maps into the parameter for the other. More specifically, problem L reduces to problem L' if there is a mapping R from instances of L to instances of L' such that (i) (I,k) is a yes-instance of L if and only if $(I',k') = R(I,k)$ is a yes-instance of L', (ii) $k' = g(k)$ for a computable function g, and (iii) R can be computed in time $O(f(k)\|I\|^c)$ where f is a computable function and c is a constant.

The *Weft Hierarchy* consists of parameterized complexity classes $\mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \cdots$ which are defined as the closure of certain parameterized problems under parameterized reductions (see [2,4,12] for definitions). There is strong theoretical evidence that parameterized problems that are hard for classes $\mathrm{W}[i]$ are not fixed-parameter tractable. For example $\mathrm{FPT} = \mathrm{W}[1]$ implies that the Exponential Time Hypothesis (ETH) fails; that is, $\mathrm{FPT} = \mathrm{W}[1]$ implies the existence of a $2^{o(n)}$ algorithm for n-variable 3SAT [4,7].

We establish our hardness results by parameterized reductions from the following parameterized decision problems (k denotes the parameter).

INDEPENDENT SET
Instance: A graph $G = (V, E)$, a non-negative integer k.
Question: Is there a set $I \subseteq V$ of size k such that for no edge $uv \in E$ we have both $u \in I$ and $v \in I$? (I is an *independent set* of G.)
Remark: This problem is W[1]-complete, see [2].

HITTING SET
Instance: Finite sets $S_1, \ldots, S_m$, a non-negative integer k.
Question: Is there a set $H \subseteq \bigcup_{i=1}^{m} S_i$ of size at most k such that $H \cap S_i \neq \emptyset$ for all $1 \leq i \leq m$? (H is a *hitting set* of $S_1, \ldots, S_m$.)
Remark: This problem is W[2]-complete, see [2].

PARTITIONED CLIQUE
Instance: A k-partite graph $G = (V, E)$ with partition $V_1, \ldots, V_k$ such that $|V_i| = |V_j|$ for $1 \leq i < j \leq k$.
Question: Are there k vertices $v_1, \ldots, v_k$ such that $v_i \in V_i$ for $1 \leq i \leq k$ and $v_i v_j \in E$ for $1 \leq i < j \leq k$? (The graph $K = (\{v_1, \ldots, v_k\}, \{\, v_i v_j : 1 \leq i < j \leq k \,\})$ is a *clique* of G.)
Remark: This problem is W[1]-complete, see [13].

3 W-Hardness

Theorem 1. *k-FLIP MAX SAT is W[1]-hard and remains W[1]-hard for 2-CNF formulas.*

Proof. We devise a parameterized reduction from INDEPENDENT SET; let (G, k) with $G = (V, E)$ be an instance of this problem. We denote the degree of a vertex $v \in V$ in G by $d(v)$ and we let $\triangle = \max_{v \in V} d(v)$; furthermore we put $m = |E|$. We construct a CNF formula F as follows. The variables of F are the vertices of G plus new variables $a_1, \ldots, a_{\triangle - 1}$, $b_1, \ldots, b_{k-1}$, $c_1, \ldots, c_m$, and z.

We define the clauses of F in five groups.

1. For each edge $uv \in E$ we introduce the clause $\{u, v\}$.
2. For each $v \in V$ and $1 \leq i \leq d(v) - 1$ we introduce the clause $\{\neg v, a_i\}$.
3. For each $1 \leq i \leq k - 1$ we introduce the clause $\{\neg z, b_i\}$.
4. For each $v \in V$ we introduce the clause $\{\neg v, z\}$.
5. For each $1 \leq i \leq \triangle - 1$, $1 \leq i' \leq k - 1$, and $1 \leq j \leq m$ we introduce the clauses $\{\neg a_i, c_j\}$, $\{\neg a_i, \neg c_j\}$, $\{\neg b_{i'}, c_j\}$, and $\{\neg b_{i'}, \neg c_j\}$.

We denote the set of clauses introduced in step i by F_i, $1 \leq i \leq 5$. Setting $F = \bigcup_{i=1}^{5} F_i$ completes the construction of F. Clearly F can be constructed in polynomial time in terms of the size of G.

Let $\tau : \mathrm{var}(F) \to \{0\}$ be the all-0-assignment of F. Observe that τ satisfies all clauses of F except for the clauses in F_1; thus $\mathrm{sat}(\tau, F) = |F| - |E|$.

Claim: G has an independent set of size k if and only if F has a truth assignment τ' such that $\mathrm{dist}(\tau, \tau') \leq k + 1$ and $\mathrm{sat}(\tau', F) > \mathrm{sat}(\tau, F)$.

We sketch the proof of the claim. Let I be an independent set of G with $|I| = k$. We define a truth assignment $\tau' : \mathrm{var}(F) \to \{0,1\}$ by setting $\tau'(x) = 1$ if $x \in I \cup \{z\}$ and $\tau'(x) = 0$ otherwise. By construction we have $\mathrm{dist}(\tau, \tau') = k + 1$; it is easy to verify that $\mathrm{sat}(\tau', F) = \mathrm{sat}(\tau, F) + 1$, thus one direction of the claim holds. Conversely, let τ' be a truth assignment of F with $\mathrm{dist}(\tau, \tau') \leq k + 1$ and $\mathrm{sat}(\tau', F) > \mathrm{sat}(\tau, F)$. Clearly $\tau(a_i) = 0$ for all $1 \leq i \leq \triangle - 1$ and $\tau(b_i) = 0$ for all $1 \leq i \leq k - 1$ since otherwise at least m clauses of F_5 would not be satisfied, a deficit that cannot be compensated elsewhere. By setting $\tau'(v) = 1$ for a single $v \in V$ we can increase the total number of satisfied clauses at most by one, and this is exactly the case if $\tau'(z) = 1$ and no clause of F_1 that contains v is already satisfied. On the other hand, $\tau'(z) = 1$ implies that all $k - 1$ clauses in F_3 are not satisfied by τ'. Therefore, the only possibility for τ' is to set z and exactly k independent vertices of V to 1. In other words, G must have an independent set of size k. Hence the claim is shown true.

We conclude that our construction provides indeed a parameterized reduction from INDEPENDENT SET to k-FLIP MAX SAT by mapping the instance (G, k) of the former problem to the instance $(F, \tau, k + 1)$ of the latter. □

Theorem 2. k-FLIP SAT *is* W[2]*-hard.*

Proof. The result follows easily by a reduction from HITTING SET. Let (H, k) be an instance of HITTING SET with $H = \{S_1, \ldots, S_m\}$ and $X = \bigcup_{i=1}^{m} S_i$. We consider H as a positive CNF formula and let $\tau : X \to \{0\}$ be the all-0-assignment on X. It is evident that H has a satisfying truth assignment $\tau' : X \to \{0,1\}$ such that $\mathrm{dist}(\tau, \tau') \leq k$ if and only if H has a hitting set of size at most k. □

Theorem 3. *The problems* k-FLIP SAT *and* k-FLIP MAX SAT *remain* W[1]*-hard if each variable occurs in at most* 3 *clauses.*

Proof. We devise a parameterized reduction from PARTITIONED CLIQUE; let $G = (V, E)$ with partition $V_1, \ldots, V_k$, $|V_1| = \cdots = |V_k| = n$, be an instance of this problem. We construct a CNF formula F. The variables of F are the vertices and edges of G plus a new variable z; we define the clauses of F as follows:

1. We introduce the clause $\{z\}$.
2. For each $1 \leq i \leq k$ we introduce the clause $C_i = V_i \cup \{\neg z\}$.
3. For each $v \in V_i$, $1 \leq i \leq k$, and each $j \in \{1, \ldots, k\} \setminus \{i\}$, we add the clause $C_{i,j,v} = \{\neg v\} \cup \{\, vu : u \in V_j \text{ and } vu \in E \,\}$.

This completes the construction of F.

Let $\tau : \mathrm{var}(F) \to \{0\}$ be the all-0-assignment of F. Observe that τ satisfies all clauses of F except clause $\{z\}$. Increasing the number of satisfied clauses is equivalent to satisfying all clauses of F, thus solutions to SAT and MAX SAT coincide for (F, τ).

Let $k' = k + \binom{k}{2} + 1$.

Claim: G contains a clique on k vertices if and only if F is satisfied by a truth assignment $\tau' : \mathrm{var}(F) \to \{0,1\}$ with $\mathrm{dist}(\tau, \tau') \leq k'$.

Let $K = (V', E')$ with $V' = \{v_1, \ldots, v_k\}$ and $v_i \in V_i$, $1 \leq i \leq k$, be a clique of G. Let τ' be the truth assignment that sets all variables in $V' \cup E' \cup \{z\}$ to 1 and all other variables to 0. It is easy to verify that $\mathrm{dist}(\tau, \tau') = k'$ and τ' satisfies F. Conversely,

let $\tau' : \mathrm{var}(F) \to \{0,1\}$ be a truth assignment that satisfies F with $\mathrm{dist}(\tau,\tau') \leq k'$. Because of the clause $\{z\} \in F$ clearly $\tau'(z) = 1$. Because of the clauses C_i it follows that each set V_i, $1 \leq i \leq k$, must contain some variable v_i with $\tau'(v_i) = 1$. Hence there is a set $V' = \{v_1, \ldots, v_k\}$, with $v_i \in V_i$ and $\tau'(v_i) = 1$ for $1 \leq i \leq k$. Let $E' = \{\, e \in E : \tau'(e) = 1 \,\}$. Since τ' sets at most k' variables to 1, and among these variables are $v_1, \ldots, v_k$ and z, we conclude that $|E'| \leq k' - k - 1 = \binom{k}{2}$. Because of the clauses C_{i,j,v_i} it follows that for each v_i and each $j \in \{1,\ldots,k\} \setminus \{i\}$ there is an edge $v_i u_j \in E'$ for some $u_j \in V_j$. Since $|E'| \leq \binom{k}{2}$ it follows that $u_j = v_j$. Hence $E' = \{\, v_i v_j : 1 \leq i < j \leq k \,\}$ and $|E'| = \binom{k}{2}$; thus $K = (V', E')$ is indeed a clique of G with k vertices. This completes the proof of the claim.

We conclude that the above construction specifies a parameterized reduction from Partitioned Clique to k-Flip (Max) Sat by mapping an instance (G,k) of the former problem to the instance (F,τ,k') of the latter.

Next we outline how the reduction can be modified so that each variable occurs in at most three clauses.

Let F^* be the CNF formula obtained from F by replacing variables of F that occur in more than three clauses by new variables (a separate variable per clause), and by adding "implication cycles" of binary clauses that ensure that new variables of F^* representing the same variable of F will have the same truth value under any satisfying assignment of F^*. Note that the all-0-assignment σ of F^* satisfies all but one clause of F^*. It is easy to verify that satisfying truth assignments of F that set exactly k' variables to 1 are in a one-to-one correspondence with satisfying truth assignments of F^* that set exactly $k^* = k^2 + \binom{k}{2} + k + 1$ variables to 1. Hence we can map the instance (G,k) of Partitioned Clique to the instance (F^*,σ,k^*) of k-Flip (Max) Sat where each variable occurs in at most three clauses. Thus the theorem follows. □

The CNF formulas F and F^* as constructed in the proof of Theorem 3 are *anti-Horn* (each clause contains at most one negative literal). We can give a dual reduction that produces *Horn* formulas (each clause contains at most one positive literal). Hence Theorem 3 remains valid for Horn and for anti-Horn formulas.

4 Fixed-Parameter Tractability

Theorem 4. *Let q be an arbitrary but fixed positive integer. k-Flip Sat is fixed-parameter tractable for q-CNF formulas.*

This result follows by a straightforward application of the bounded search tree method [2]; branching on literals of unsatisfied clauses gives a search tree with $O(q^3)$ nodes.

Theorem 5. *Let p, q be arbitrary but fixed positive integers. k-Flip Max Sat is fixed-parameter tractable for q-CNF formulas where each variable occurs in at most p clauses.*

The proof of this theorem requires some preparation. Let S denote a finite relational structure and φ a first-order (FO) formula. S is a *model* of φ (in symbols $S \models \varphi$) if φ is true in S in the usual sense (see, e.g., [4]). We consider the following problem, parameterized by the length of the considered FO formula φ.

FO MODEL CHECKING
Instance: A finite structure S, a FO formula φ.
Question: Does $S \models \varphi$ hold?

We associate with a relational structure S its *Gaifman graph* $G(S)$, whose vertices are the elements of the universe of S, and where two distinct vertices are joined by an edge if and only if they occur together in some tuple of a relation of S. By means of Gaifman graphs one can associate graph invariants such as maximum degree or treewidth with a relational structure. Frick and Grohe [5] have shown that FO MODEL CHECKING for structures of bounded maximum degree is fixed-parameter tractable (they show the result for classes of structures of "effectively bounded local treewidth" which includes structures of bounded maximum degree as a special case). In fact, Frick and Grohe's result establishes that FO MODEL CHECKING is fixed-parameter tractable for the combined parameter $d + k$ where d bounds the maximum degree of the given structure and k bounds the length of the FO formula (see [4] for more background and technical details).

Proof of Theorem 5. Consider an instance (F, τ, k) of k-FLIP MAX SAT where each clause contains at most q literals and each variable occurs in at most p clauses.

We represent the pair (F, τ) by a relational structure S_F as follows. For every variable x of F and every clause C of F, the universe of S_F contains distinct elements a_x and a_C, respectively. The relations of S_F are defined as follows.

$$
\begin{aligned}
V &= \{\, a_x : x \in \mathrm{var}(F) \,\} \text{ (variables)} \\
C &= \{\, a_C : C \in F \,\} \text{ (clauses)} \\
P &= \{\, (a_x, a_C) : x \in \mathrm{var}(F),\ C \in F,\ \text{and } x \in C \,\} \text{ (positive occurrence)} \\
N &= \{\, (a_x, a_C) : x \in \mathrm{var}(F),\ C \in F,\ \text{and } \neg x \in C \,\} \text{ (negative occurrence)} \\
T &= \{\, a_x : x \in \mathrm{var}(F) \text{ and } \tau(x) = 1 \,\} \text{ (variables that are true under } \tau)
\end{aligned}
$$

The maximum degree of S_F is bounded by the maximum of p and q.

For fixed t, s, u it is not difficult to construct a FO formula $\varphi_{t,s,u}$ which states that there exist t distinct variables $x_1, \ldots, x_t$ and $s+u$ distinct clauses $y_1, \ldots, y_s, z_1, \ldots, z_u$ such that the clauses y_j are exactly those which are not satisfied by τ but are satisfied after flipping the truth values of the variables x_i, and the clauses z_j are exactly those which are satisfied by τ but are not satisfied after the flipping. We define φ as the disjunction of various instances of $\varphi_{t,s,u}$ with $1 \leq t \leq k$ and $0 \leq u < s \leq p \cdot t$. Clearly the length of φ depends on k and p only, and it holds that $S_F \models \varphi$ if and only if (F, τ, k) is a yes-instance of k-FLIP MAX SAT. Consequently Theorem 5 follows by Frick and Grohe's result. □

The proofs of Theorems 4 and 5 show that the considered problems are even fixed-parameter tractable if p and q are part of the parameter and not constants. That is, k-FLIP SAT is fixed-parameter tractable for parameter $k + p$ and k-FLIP MAX SAT is fixed-parameter tractable for parameter $k + p + q$.

References

1. Dantsin, E., Goerdt, A., Hirsch, E.A., Kannan, R., Kleinberg, J.M., Papadimitriou, C.H., Raghavan, P., Schöning, U.: A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. Theoret. Comput. Sci. 289(1), 69–83 (2002)
2. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Monographs in Computer Science. Springer, Heidelberg (1999)
3. Fellows, M.R.: Blow-Ups, Win/Win's, and Crown Rules: Some New Directions in FPT. In: Bodlaender, H.L. (ed.) WG 2003. LNCS, vol. 2880, pp. 1–12. Springer, Heidelberg (2003)
4. Flum, J., Grohe, M.: Parameterized Complexity Theory. Texts in Theoretical Computer Science. An EATCS Series, vol. XIV. Springer, Heidelberg (2006)
5. Frick, M., Grohe, M.: Deciding first-order properties of locally tree-decomposable structures. J. ACM 48(6), 1184–1206 (2001)
6. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Elsevier/Morgan Kaufmann (2004)
7. Impagliazzo, R., Paturi, R., Zane, F.: Which problems have strongly exponential complexity? J. of Computer and System Sciences 63(4), 512–530 (2001)
8. Johnson, D.S., Papadimitriou, C.H., Yannakakis, M.: How easy is local search? J. of Computer and System Sciences 37(1), 79–100 (1988)
9. Khuller, S., Bhatia, R., Pless, R.: On local search and placement of meters in networks. SIAM J. Comput. 32(2), 470–487 (2003)
10. Krokhin, A.A., Marx, D.: On the hardness of losing weight. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 662–673. Springer, Heidelberg (2008)
11. Marx, D.: Searching the k-change neighborhood for TSP is W[1]-hard. Oper. Res. Lett. 36(1), 31–36 (2008)
12. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Lecture Series in Mathematics and its Applications. Oxford University Press, Oxford (2006)
13. Pietrzak, K.: On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. J. of Computer and System Sciences 67(4), 757–771 (2003)
14. Samer, M., Szeider, S.: Fixed-parameter tractability. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, ch. 13, pp. 425–454. IOS Press, Amsterdam (2009)
15. Yagiura, M., Ibaraki, T.: Analyses on the 2 and 3-flip neighborhoods for the MAX SAT. J. Comb. Optim. 3(1), 95–114 (1999)

A Novel Approach to Combine a SLS- and a DPLL-Solver for the Satisfiability Problem

Adrian Balint, Michael Henn, and Oliver Gableske*

Ulm University
Institute of Theoretical Computer Science
89069 Ulm, Germany
{adrian.balint,michael.henn,oliver.gableske}@uni-ulm.de

Abstract. The paper at hand presents a novel and generic approach on how to combine a SLS and a DPLL solver to create an incomplete hybrid SAT solver. In our approach, the SLS solver gets supported by a DPLL solver to boost its performance. In order to develop the idea behind our approach, we first define the term of a search space partition (SSP) and explain its construction and use. For testing our new approach, which utilizes SSPs, we implemented it in the solver *hybridGM*, using *gNovelty+* and *March_ks*. After explaining the implementation details, we perform an empirical study on several publicly available benchmarks in order to test the performance of the new hybrid SAT solver. The results indicate a superior performance of *hybridGM* over *gNovelty+*, proving our new approach to be worthwhile.

1 Introduction

The propositional satisfiability problem (SAT) is one of the most studied $\mathcal{NP}$-complete problems [1] in computer science. One reason for that is the wide range of SAT's practical applications ranging from hardware verification to planning and scheduling.

Most of today's SAT solvers' architectures are based on one of the following paradigms: DPLL (based on DP [4]) and SLS (based on GSAT [19]). Both paradigms have quite oppositional benefits and drawbacks. SLS solvers scale very well on random instances and use comparatively little memory. On the other hand they can not disclose the unsatisfiability of a problem. In contrast to that, DPLL solvers are good at solving industrial and structured problems and they can ascertain if a problem is satisfiable or unsatisfiable, but they have difficulties solving random instances and use a larger amount of memory than SLS solvers.

Altogether, DPLL and SLS solvers seem to complement each other very well, and therefore, the idea of combining both approaches seems promising. The ideal hybrid SAT solver, that follows both paradigms, would be very fast, scale very well, and would be complete. This is why the construction of hybrid SAT solvers is an active field of research, that has seen large efforts over the past years.

* Funded by the Graduate School *Mathematical Analysis of Evolution, Information and Complexity* at Ulm University.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 284–297, 2009.

2 Related Work

Considerable effort has been undertaken to create hybrid SAT solvers for more than a decade now. In general, three different approaches to create such hybrid SAT solvers have emerged.

First, one tries to use a SLS solver to support a DPLL solver ([2,17,5,10,7,8]). Such a support can come in various ways. In [2], a SLS solver is used to derive weights for clauses. These clause weights are then used by a DPLL solver to preferably branch on variables that occur more often in clauses with higher weights. In [17], a SLS solver is used to find inconsistent kernels in a formula. This knowledge is then used to narrow the search of a DPLL solver to only the inconsistent part of a formula. As a result, the global unsatisfiability of a formula can be shown in less computation time. In [5], a SLS solver is used to determine an ordering of the branching variables that a DPLL solver should follow. In [8], a SLS solver is used to identify areas of the search space that are more likely to contain a solution. These areas are then represented as partial assignments, that a DPLL solver starts its search with. Confined to such a portion of the search space, the DPLL solver is usually faster in finding a solution than it would be when left to itself.

As a second approach, one might be able to use information gathered by DPLL solvers on a certain formula to support the search of a SLS solver ([14,6,9]). In [6], a DPLL solver is called whenever the SLS solver has moved into a local minimum in the search space. The approach adds implied clauses (learned by the DPLL solver) to modify the search space landscape the SLS solver works in. This learning process is repeated until the SLS solver is able to move out of the (former) local minimum. In [9], a DPLL solver will derive implications between variables when arriving in a certain node of its search tree. With these implications, a reduced version of the currently investigated formula is created, consisting of only equivalence classes. When applying the SLS solver on the reduced formula, it will consider the equivalence classes rather than the original variables. This in turn helps the SLS solver to concentrate on actually different variables, when making its choice what variable is to be flipped next. The SLS solver is then allowed to perform a maximum number of flips for searching a solution (under the DPLL provided preconditions). If the SLS solver finds a solution during its search, the algorithm terminates. If the SLS solver used up its allowed flips and did not find a solution, the DPLL solver comes back into play and continues traveling down its search tree.

The third approach on creating hybrid SAT solvers is peer-like, where SLS and DPLL solvers are supposed to benefit equally from each other as presented in [7,15]. The *hbisat* solver [7] and its successor *hinotos* [15] both use a SLS solver that first tries to solve the formula. When a certain criterion is met (e.g. only a certain number of unsatisfied clauses remain), a DPLL solver is called, that is supposed to solve these clauses separately. When the DPLL solver finds a model for this partial set of clauses, it will return the corresponding assignment to the SLS solver, that will then use this assignment to continue its search. Eventually, the SLS solver will be able to find a solution. If this is not the case, the set

of clauses that the DPLL solver investigates, grows over time. Eventually, the DPLL solver will be provided with enough clauses to find a contradiction and deduce the unsatisfiability of the formula or it can provide a solution for the complete formula.

Despite the efforts undertaken so far, no truly superior hybrid SAT solver has yet emerged. This is why we think that more effort is needed in this field of research. This paper is supposed to contribute to these efforts.

3 Preliminary Study

Towards the development of a hybrid SAT solver we started by investigating the search of a SLS solver, looking for weaknesses. The main goal was to find a way to support a SLS solver by additionally using a DPLL solver to overcome these weaknesses and thereby boost the SLS solver's performance.

Looking at the results of the SLS solvers in the random category of the SAT 2007 Competition, we noticed that the runtime of a SLS solver on formulae of the same size can vary greatly. To analyze this effect we focused our attention on the winning SLS solver of the SAT 2007 Competition in the random category, *gNovelty+* [20,18]. Our assumption was that the search space structure of the hard to solve formulae contained many attractive local minima that were visited by the SLS solver very often. To verify this assumption we tried to cluster all points from *gNovelty+*'s search trajectory. This approach was unsuccessful because of the too huge amount of data that had to be clustered. Another approach was to analyze only the points where the objective function had very low values (i.e. local minima and their close neighborhood). Because the amount of data was still too large, we used a bloom filter to save all local minima. Then we checked how many assignments, that *gNovelty+* visited, fell in the neighborhood of the saved local minima. The maximum matching we could reach was 2%. This indicated that the diversification of *gNovelty+*'s search is excellent.

The next thing to analyze was if the intensification of *gNovelty+*'s search around these local minima was good enough to assure with a high probability that there are no solutions. To prove this we would have had to search the complete neighborhood of a local minimum within a certain Hamming distance. This is possible for small formulae, but for the formulae having thousands of variables the neighborhood is far too large to be computed in foreseeable time. Zhang showed in [23] that the Hamming distance between a good local minimum and the nearest solution is correlated with the quality of that local minimum. Attempting to find such a correlation we came up with the term of a "search space partition". To formalize this we have to give some definitions.

3.1 Search Space Partition (SSP)

Let F be a CNF formula containing n variables $\{x_1, x_2, \ldots, x_n\}$, $x_i \in \mathbb{B}$ where $\mathbb{B} = \{0, 1\}$. A complete assignment of the formula F is $\alpha \in \mathbb{B}^n$. The application of the assignment α on formula F is denoted by $F(\alpha) \in \{0, 1\}$. A partial assignment

is $\beta \in \mathbb{RB}^n$ where $\mathbb{RB} = \{0, 1, ?\}$. The application of a partial assignment β on F is denoted by $F(\beta) = F'$, where F' is a new formula in which the value of $\beta[i]$ is assigned to x_i if $\beta[i] \in \{0, 1\}$ and x_i remains unchanged if $\beta[i] =? \; \forall i \in \{1, \ldots, n\}$. Additionally, we assume that the obvious simplifications have been applied to $F(\beta) = F'$. The number of ?-symbols in β (size of β) is described by $|\beta|_?$, representing the number of remaining variables in F'.

Example 1. For

$$F = (x_1 \vee \overline{x_2} \vee x_5) \wedge (\overline{x_1} \vee x_3 \vee \overline{x_5}) \wedge (\overline{x_3} \vee x_4 \vee x_5) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_5})$$

an example for α and β could be $\alpha = (1, 0, 1, 1, 0)$ and $\beta = (?, 1, 1, ?, ?)$. The application of α and β on F is: $F(\alpha) = 1$ and $F' = F(\beta) = (x_1 \vee x_5) \wedge (x_4 \vee x_5) \wedge (\overline{x_1} \vee \overline{x_5})$.

Definition 1 (flip trajectory of a SLS solver). *Given a SLS solver S with input formula F and a complete starting assignment α_s we define the flip trajectory of $S(F, \alpha_s)$ as $T_S(F, \alpha_s) = (t_1, \ldots, t_w)$ where $t_i \in \{x_1, \ldots, x_n\}$ denote the variables being flipped by the SLS-algorithm S, and w is the total number of flips made, starting with the formula F and the initial assignment α_s.*

Example 2. Given the formula from example 1 and a starting assignment $\alpha_s = (0, 1, 0, 0, 0)$, a possible flip trajectory that would lead to a satisfying assignment could be $T_S(F, \alpha_s) = (x_1, x_5, x_3, x_2)$

Definition 2 (search space partition). *We define a search space partition (SSP) by construction: Given a complete assignment α_j, which was visited by S in the j'th flip of the trajectory, we construct the SSP by starting with $k = 0$ and $\beta = \alpha_j$. Then we repeat setting $\beta[t_{j+k}] =?$ and $\beta[t_{j-k}] =?$, where $t_{j \pm k} \in T_S(F, \alpha_s)$, and increasing k by 1 until $|\beta|_? \geq c \cdot n$ where c is some constant $c \in (0, 1)$ (to be determined later).*

Example 3. Let $\alpha_7 = (0, 0, 1, 1, 0, 1, 0, 1, 1, 1)$ be a complete assignment for a formula F with 10 variables and let the surrounding flip trajectory be

$$T_S(F, \alpha_s) = (x_2, x_6, x_1, x_9, x_1, x_6, \underline{\mathbf{x_1}}, x_3, x_9, x_1, x_1, x_8, x_3, \ldots)$$

If we set $c = 0.5$ and start to construct a SSP from position $j = 7$ in $T_S(F, \alpha_s)$, then the first variable that is unassigned in β is x_1 ($k = 0$). In the next step x_3 and x_6 get unassigned ($k = 1$) according to $T_S(F, \alpha_s)$. This procedure is repeated until $|\beta|_? \geq 5$. After five steps the process will stop with $\beta = (?, 0, ?, 1, 0, ?, 0, ?, ?, 1)$.

The connection between an assignment and a SSP is as follows. A complete assignment is a point in the space $\mathbb{B}^n$, whereas a SSP is a set of points (subset of $\mathbb{B}^n$) of dimension $\lceil c \cdot n \rceil$, $c \in (0, 1)$, that can be characterized by a partial assignment. Intuitively, the SSP created with the help of a SLS solver can also be seen as the confidence of the solver, that certain values of its assignment are set correctly.

The notation of search space partitions was also used by Wu and Hsiao in [22], where they propose a simulation-based algorithm for checking the safety property of digital systems. Although the notion is similar, the concepts are quite different. The search space in [22] is represented by the internal nodes of a Boolean circuit. These nodes are joined together to node sets that represent the search space partition. Because the nodes contained in a node set are strongly connected with each other by gates, some value combinations can be dismissed from the search space partition, so that the whole search space gets smaller. In order to point out the difference to our work, we provide the following example.

Example 4. Suppose we have a circuit with nine internal nodes denoted by $\{n_1, \ldots, n_9\}$ and connected by gates without flip-flops. A possible partitioning in sets could be: $\{n_1, n_2, n_3, n_4\}$, $\{n_7, n_8\}$ and $\{n_5, n_6, n_9\}$. Now suppose that the nodes n_1 and n_2 are the inputs of an and-gate, so are n_3 and n_4. Then from all possible $2^4 = 16$ inputs of the subset only 9 are valid ones, so that 7 are dismissed. Now suppose that for the second set only 3 possible value combinations are valid and for the third one only 5. The whole search space now contains only $9 \cdot 3 \cdot 5 = 135$ possible inputs out of 512.

We instead construct a SSP (which is a partial assignment) by starting from a complete assignment and then unassigning variables. In our case unassigning 4 variables for an assignment of $\{n_1, \ldots, n_9\}$ would lead to a SSP containing 16 complete assignments.

4 Explaining the Construction and Use of Search Space Partitions

As mentioned in section 3 we are interested in the study of local minima and their neighborhood. Therefore the α_j mentioned in the definition of a SSP would ideally be a local minimum. A SSP created as defined above overlaps with the Hamming neighborhood of the local minimum but is not the same.

Figure 1 describes how SSPs are created around local minima with exactly one unsatisfied clause. It has to be mentioned that a SSP can contain multiple minima.

A local minimum exists, because several variables imply a complex conflict that cannot be solved by a SLS solver within the next flip. To resolve such a conflict, the SLS solver would have to flip several of these variables. The variables that are flipped are determined by the variable selection heuristic of the SLS solver, which is usually quite reasonable. However, the order and number in which the variables are flipped (and therefore, the resulting assignments) are usually determined by the landscape of the objective function. This landscape will, however, not always guide the right way. Therefore, we can identify probably conflicting variables in a local minimum by monitoring the flips made by the SLS solver around the discovered local minimum in the trajectory. Their correct values are then to be determined.

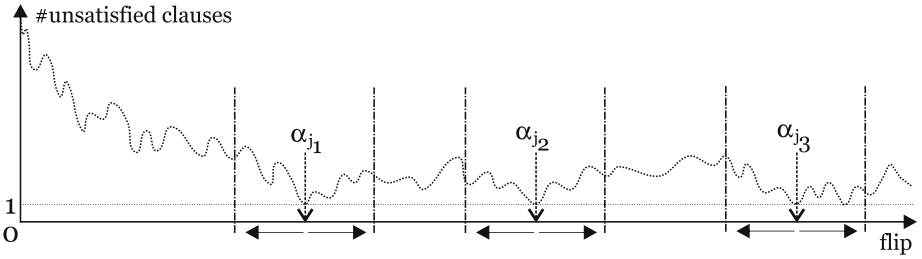


Fig. 1. A schematic visual of the objective function and the flips used for the construction of a SSP

Using this knowledge, we can create a partial assignment by unassigning these identified variables in the complete assignment of the local minimum, what leads to a SSP. Given such a SSP, our hybrid strategy consists in calling a DPLL solver, which will try to resolve the conflict by finding an assignment for the remaining variables in the SSP. In case the DPLL solver finds an assignment that resolves the conflict using the unassigned variables, a solution for the complete formula is found. In case such an assignment does not exist, the conflict can not be resolved by using only the free variables in the SSP. In this case, the search of the SLS solver must continue to identify a new local minimum.

All in all, a generic algorithm implementing the above idea would use a SLS solver to localize good local minima, build a SSP, apply the partial assignment of the SSP on the formula and try to find a solution for the simplified formula with a DPLL solver. This process would be repeated until a solution is found or until another stopping criterion is met. The algorithm can not prove the unsatisfiability of the problem but it could speed up the SLS solver by finding a solution sooner.

5 hybridGM

To check if our approach is promising we implemented a hybrid SAT solver which we call *hybridGM*. The SLS solver used is *gNovelty+*. Because we expected that a SSP does not contain a solution in the majority of cases (which we found to be true), we needed a solver that can prove the unsatisfiability of a given (sub-)formula fast and because *March_ks* was the winner of *random UNSAT* category of the SAT 2007 Competition [20], we have chosen *March_ks* for the DPLL component of *hybridGM*.

When implementing *hybridGM* two questions arose. First, which local minima should be used to create a SSP? Using every appearing local minimum in the objective function leads to an overwhelming workload. Numerous local minima are discovered by the SLS solver as we have noticed when analyzing the search trajectory of *gNovelty+*. To reduce the workload, we confine ourselves to using only those assignments that leave exactly one clause in the formula unsatisfied.

Second, how large should the SSP be, i.e. how to set c, where $|\beta|_? \geq c \cdot n$? The more variables in β are set to ?, the higher is the probability to resolve all

conflicts in the corresponding SSP. On the other hand, the less variables in β are set to ?, the faster the DPLL solver can return a result. Therefore, we have to compromise between the probability to resolve all conflicts and the runtime of the DPLL solver.

For initial tests we have chosen $|\beta|_? = 0.5 \cdot n$. In this case, the DPLL solver almost instantaneously returns SAT/UNSAT. If it reports UNSAT together with a unary conflict, we increase c by 0.05.

Listing 1. Pseudocode for hybridGM

```
INPUT: formula F, cutoff. OUTPUT: model for F or UNKNOWN.
hybridGM(F, cutoff){
    α = α_s = randomly generated starting assignment;
    numFlips = 0; c = 0.5; barrier= 1; collectSSP = FALSE;
    while(numFlips < cutoff){
        var = pickVar();
        append(T_gNovelty+(F, α_s),var);
        α[var] = 1−α[var];
        numFlips++;
        if (α is model for F) return α;
        if (numUnsatClauses ≤ barrier){
            β = α;
            collectSSP = TRUE;
            j = numFlips; k = 0;
        }
        if (collectSSP == TRUE){
            β[ variableIndex( T_gNovelty+(F, α_s)[j + k] ) ] =?;
            β[ variableIndex( T_gNovelty+(F, α_s)[j − k] ) ] =?;
            k++;
        }
        if (|β|_? ≥ cn){
            μ = March_ks(F, β);
            if (μ is model for F) return μ;
            else if (unaryConflictOccurred() == TRUE) c = c + 0.05;
            collectSSP = FALSE;
        }
        updateParameters(); //noise, scores
    }
    return UNKNOWN;
}
```

The framework of *hybridGM* is that of *gNovelty+* with the extension of collecting SSPs when the number of unsatisfied clauses reaches a certain barrier. If this happens, the number of the flip in the trajectory will be saved in j. This j is used for the construction of the SSP as given in definition 2. When the size of the SSP has reached the size of cn then *March_ks* is called with the initial formula and the partial assignment β. If *March_ks* succeeds in solving the formula

by extending the partial assignment, the solution is returned, else if *March_ks* discovers a unary conflict, c is adapted. Finally, the parameters of *gNovelty+* are updated and the search continues until the number of flips reaches the cutoff. Before presenting the results of the empirical study we first give a short overview about the solvers *gNovelty+* and *March_ks*.

6 The SAT Solvers Used for the Hybridization

6.1 *gNovelty+*

gNovelty+ [18] is a SLS solver that follows the WalkSAT architecture, and it can be seen as an improvement of the SLS solver G^{2}WSAT [16]. In its core, *gNovelty+* utilizes a gradient-based variable score update scheme to calculate candidate variables for the next flip of the WalkSAT procedure (as G^{2}WSAT originally does). However, several improvements over G^{2}WSAT have been introduced for *gNovelty+*.

First, the original variable selection heuristic of *G^2 WSAT*, called Novelty++, was replaced by AdaptNovelty+ [12]. Second, *gNovelty+* applies a clause weighting scheme quite similar to *SAPS* (see [13]) but with purely additive weight update functions [18].

gNovelty+ has proven to be very competitive during the SAT 2007 Competition when it comes to satisfiable uniform random k-SAT (see [18,20] for more details). Therefore, it is considered a state-of-the-art SLS SAT solver.

6.2 *March_ks*

March_ks [11] is a double look-ahead DPLL solver, that utilizes numerous features and performed very well at the SAT 2007 Competition [20]. Originating from a solver called *march*, several versions have evolved over time. The first solver in this line with reasonable performance was *March_eq*.

March_eq introduced the equivalency reasoning into the march solver family, enabling it to find and separately exploit variable equivalences in a searched instance. *March_dl* is its successor.

March_dl introduced the double look-ahead into the march solver family, enabling *March_dl* to detect failed literals sooner than *March_eq*. Furthermore, *March_dl* introduced a new branching strategy called local branching. The successor of *March_dl* was *March_ks* [11].

March_ks introduced two new features to the march solver family. First, it used a modified version of the double look-ahead, called adaptive double look-ahead [11]. Second, *March_ks* makes use of a new backtracking strategy called distribution jumping.

March_ks has proven to be very competitive when it comes to uniform random k-SAT formulae, and is therefore considered one of todays state-of-the-art DPLL SAT solvers. For more details on the performance of *March_ks* see [20].

7 Empirical Study

For the implementation of *hybridGM* we used the *gNovelty+* code, with a changed smoothing probability from 0.4 to 0.33. The barrier was fixed to one, so that SSPs are built only when exactly one clause remains unsatisfied. In some cases when calling *March_ks* and returning the result to *gNovelty+* we observed that a part of the memory allocated by *March_ks* data structure is not released. This memory leak is very small but if the number of *March_ks* calls gets too large, we get an out of memory exception. To avoid this case we limited the number of *March_ks* calls to 5000. Furthermore, we sometimes get a signal 6 abort, which indicates that there are further bugs to be resolved in the code.

We also tested different settings for the border $|\beta|_?$, i.e. the number of variables in a SSP, that must be unassigned before *March_ks* examines it. However, we noticed that an initial value $> 0.5 \cdot n$ just raises the calculation times of *March_ks*, without improving its success rate of resolving all conflicts in a SSP.

7.1 Soft- and Hardware

The *gNovelty+* and the *adaptg2wsat0* code we used for the comparison was the one submitted to the SAT 2007 Competition[1]. The *March_ks* code we used was a bug-fixed version of the SAT 2007 Competition[2]. The code of *hybridGM* can be downloaded from our website at `http://www.uni-ulm.de/in/theo/research/sat-solving.html`.

The solvers were run on a part of the bwGrid, where we were provided with a total of 140 Intel Harpertown quad-core CPUs with 2.83 GHz and 8 GByte RAM. The operating system was Scientific Linux.

7.2 The Benchmark Formulae

We used formulae from two benchmark sets to test *hybridGM* against *gNovelty+* and *adaptg2wsat0*: the SAT 2007 Competition benchmark [20], and the SATLIB benchmark [21].

Concerning the SAT 2007 Competition benchmark, we selected numerous satisfiable random and industrial formulae. First, we randomly picked 24 of the satisfiable $2 + p$ formulae of different sizes. Second, we randomly picked 5 small uniform random 3-SAT formulae with 650 variables. Third, we picked numerous $3-$SAT formulae from the large-size category, as well as some $5-$, and $7-$SAT formulae from the on-threshold category. Concerning the SAT 2007 Competition benchmark, we also ran tests on some industrial and crafted instances that *gNovelty+* was able to solve during the SAT 2007 Competition. Concerning the SATLIB benchmark, we chose some prominent crafted and industrial instances.

7.3 Results

When taking a closer look at table 1, one discoveres that *hybridGM* dominated *gNovelty+* in several areas. One of these areas is the $2 + p$ formula set of the

[1] `http://www.satcompetition.org/2007/winners.tgz`
[2] `http://www.st.ewi.tudelft.nl/sat/Sources/sat2007/march_ks.zip`

Table 1. The table presents the results of our empirical study. Each solver performed 100 runs per instance (*gNovelty+* and *hybridGM* were started with the same seed). Runtimes are given in seconds: mean | median. If the solver was not able to succeed 100 times (each within 2000 seconds), we just give the success rate for the instance in percent. The *hybridGM* column also contains a tuple, that presents which component found the solution how often within the 100 runs (*gNovelty+*, *March_ks*). The gain column represents the speed-up of *hybridGM* over *gNovelty+* (> 1 indicates that *hybridGM* was faster or had a better success rate, ≤ 1 indicates the opposite).

Instance	gNovelty+	adaptG2-WSAT0	hybridGM (gNov,March)	Gain
SAT 2007 Competition random instances				
unif2p-p0.7-v3500-c9345-S1286605994-07	91%	16.38 | 11.84	38.96 | 29.72 (20, 80)	**>1**
unif2p-p0.7-v3500-c9345-S1568322528-08	10%	2.91 | 1.63	9.63 | 7.82 (9, 91)	**>1**
unif2p-p0.7-v4500-c12015-S1545977164-16	2%	50.64 | 7.84	109.25 | 91.15 (7, 93)	**>1**
unif2p-p0.7-v4500-c12015-S1973057201-08	25%	138.52 | 42.71	121.52 | 82.44 (23, 77)	**>1**
unif2p-p0.7-v5500-c14685-S1568197186-18	2.00 | 1.39	0.39 | 0.24	0.46 | 0.42 (37, 63)	**4.35**
unif2p-p0.7-v5500-c14685-S915337037-05	95%	3.33 | 1.60	20.51 | 5.79 (24, 76)	**>1**
unif2p-p0.7-v6500-c17355-S1097641288-15	97.64 | 69.35	3.75 | 1.61	4.31 | 2.57 (20, 80)	**22.65**
unif2p-p0.7-v6500-c17355-S152598520-02	226.64 | 168.00	10.23 | 1.90	2.17 | 1.66 (27, 73)	**104.44**
unif2p-p0.8-v1295-c4027-S1679085272-10	0.74 | 0.58	6.06 | 4.20	0.45 | 0.36 (73, 27)	**1.64**
unif2p-p0.8-v1295-c4027-S1762612346-15	136.06 | 94.03	1.13 | 0.79	2.65 | 2.20 (9, 91)	**51.34**
unif2p-p0.8-v1665-c5178-S1363528912-04	20.84 | 16.16	5.11 | 3.33	5.21 | 4.65 (73, 27)	**4.00**
unif2p-p0.8-v1665-c5178-S1404069132-16	265.54 | 196.49	5.70 | 3.10	23.56 | 17.24 (13, 87)	**11.27**
unif2p-p0.8-v2035-c6328-S1703132040-12	1.47 | 1.02	0.74 | 0.32	0.46 | 0.38 (44, 56)	**3.20**
unif2p-p0.8-v2035-c6328-S316347254-19	0.75 | 0.48	0.18 | 0.14	0.23 | 0.17 (40, 60)	**3.26**
unif2p-p0.8-v2405-c7479-S1163137157-19	8.18 | 6.14	4.00 | 2.27	9.67 | 8.01 (28, 72)	0.85
unif2p-p0.8-v2405-c7479-S183991542-09	33%	33.50 | 21.10	92% (5, 87)	**>1**
unif2p-p0.9-v1170-c4235-S1575802003-16	0.94 | 0.74	0.31 | 0.24	0.41 | 0.29 (59, 41)	**2.29**
unif2p-p0.9-v1170-c4235-S2131244303-19	52.26 | 31.82	1.45 | 1.21	2.80 | 2.07 (30, 70)	**18.66**
unif2p-p0.9-v630-c2280-S1804595013-08	4.32 | 3.33	0.91 | 0.67	1.72 | 1.35 (31, 69)	**2.51**
unif2p-p0.9-v630-c2280-S2099846342-04	0.35 | 0.27	0.14 | 0.10	0.33 | 0.23 (88, 12)	**1.06**
unif2p-p0.9-v810-c2932-S1014417748-14	0.12 | 0.08	0.08 | 0.06	0.11 | 0.09 (75, 25)	**1.09**
unif2p-p0.9-v810-c2932-S1274825698-06	0.27 | 0.21	0.06 | 0.05	0.15 | 0.10 (59, 41)	**1.80**
unif2p-p0.9-v990-c3583-S361865778-09	4.55 | 3.71	2.34 | 1.49	2.37 | 1.67 (67, 33)	**1.92**
unif2p-p0.9-v990-c3583-S461590508-14	0.28 | 0.21	0.10 | 0.08	0.23 | 0.18 (56, 44)	**1.22**
unif-k3-r4.261-v650-c2769-S1089058690-02	0.25 | 0.16	0.17 | 0.10	0.38 | 0.28 (61, 39)	0.66
unif-k3-r4.261-v650-c2769-S1159448555-06	0.46 | 0.29	0.24 | 0.19	0.67 | 0.53 (76, 24)	0.69
unif-k3-r4.261-v650-c2769-S1172355929-14	0.04 | 0.02	0.02 | 0.02	0.05 | 0.04 (79, 21)	0.80
unif-k3-r4.261-v650-c2769-S1470952774-07	4.31 | 3.42	2.21 | 1.89	7.25 | 5.51 (71, 29)	0.59
unif-k3-r4.261-v650-c2769-S1481730841-18	0.07 | 0.06	0.07 | 0.04	0.09 | 0.07 (76, 24)	0.78
unif-k3-r4.2-v10000-c42000-S1173369833-06	73.87 | 50.01	11%	7.28 | 5.61 (23, 77)	**10.15**
unif-k3-r4.2-v10000-c42000-S1912540524-08	207.11 | 152.90	1%	16.85 | 14.69 (28, 72)	**12.29**
unif-k3-r4.2-v10000-c42000-S421554531-04	98.25 | 79.86	7%	11.66 | 9.52 (30, 70)	**8.43**
unif-k3-r4.2-v10000-c42000-S597645631-10	101.45 | 77.88	5%	8.01 | 6.37 (31, 69)	**12.67**
unif-k3-r4.2-v10000-c42000-S657313757-16	98%	0%	59.16 | 46.26 (33, 67)	**>1**
unif-k3-r4.2-v10000-c42000-S897388318-05	205.65 | 170.49	2%	22.51 | 19.32 (24, 76)	**9.14**
unif-k3-r4.2-v10000-c42000-S971732863-03	64.10 | 42.61	19%	7.54 | 5.85 (37, 63)	**8.50**
unif-k3-r4.2-v13000-c54600-S1054448974-13	97%	0%	44.89 | 38.74 (36, 64)	**>1**
unif-k3-r4.2-v13000-c54600-S1416986890-04	331.30 | 250.70	0%	22.02 | 17.58 (23, 77)	**15.05**
unif-k3-r4.2-v13000-c54600-S161446644-14	55%	0%	99% (33, 66)	**>1**
unif-k3-r4.2-v13000-c54600-S1890278326-03	98%	0%	35.77 | 30.03 (35, 65)	**>1**
unif-k3-r4.2-v13000-c54600-S287388441-16	300.62 | 224.63	0%	18.23 | 15.29 (26, 74)	**16.49**
unif-k3-r4.2-v16000-c67200-S1099746708-06	23%	0%	79% (24, 55)	**>1**
unif-k3-r4.2-v16000-c67200-S1415445307-13	67%	0%	127.96 | 84.56 (35, 65)	**>1**
unif-k3-r4.2-v16000-c67200-S1600965758-04	18%	0%	73% (18, 55)	**>1**
unif-k3-r4.2-v16000-c67200-S1826381479-08	550.04 | 457.84	0%	38.00 | 33.06 (23, 77)	**14.47**
unif-k3-r4.2-v16000-c67200-S1980187645-03	42%	0%	416.34 | 344.16 (33, 67)	**>1**
unif-k3-r4.2-v16000-c67200-S202413125-05	28%	0%	67% (19, 48)	**>1**
unif-k3-r4.2-v16000-c67200-S448238512-10	38%	0%	88% (22, 66)	**>1**
unif-k3-r4.2-v16000-c67200-S791125864-16	88%	0%	85.56 | 69.47 (33, 67)	**>1**

Table 1. (*continued*)

Instance	gNovelty+	adaptG2-WSAT0	hybridGM (gNov,March)	Gain
unif-k3-r4.2-v16000-c67200-S81758219-15	99%	0%	45.38 | 38.10 (27, 73)	**>1**
unif-k3-r4.2-v16000-c67200-S886048189-14	87%	0%	33.89 | 31.33 (31, 69)	**>1**
unif-k3-r4.2-v19000-c79800-S1106616038-10	74%	0%	92.02 | 70.75 (39, 61)	**>1**
unif-k3-r4.2-v19000-c79800-S1172889356-05	15%	0%	99% (27, 72)	**>1**
unif-k3-r4.2-v19000-c79800-S1299985238-16	2%	0%	60% (16, 44)	**>1**
unif-k3-r4.2-v19000-c79800-S1314701073-08	73%	0%	123.08 | 101.84 (27, 73)	**>1**
unif-k3-r4.2-v19000-c79800-S1330787624-15	3%	0%	25% (8, 17)	**>1**
unif-k3-r4.2-v19000-c79800-S1496322949-03	24%	0%	97% (35, 62)	**>1**
unif-k3-r4.2-v19000-c79800-S1661055114-06	99%	0%	39.10 | 33.75 (27, 73)	**>1**
unif-k3-r4.2-v19000-c79800-S1753083943-04	99%	0%	28.23 | 23.99 (38, 62)	**>1**
unif-k3-r4.2-v19000-c79800-S1875179522-13	470.69 | 381.00	0%	25.59 | 20.93 (25, 75)	**18.39**
unif-k3-r4.2-v19000-c79800-S49237390-14	50%	0%	129.71 | 108.21 (30, 70)	**>1**
unif-k3-r4.2-v4000-c16800-S1178874381-13	8.11 | 6.02	184.52 | 110.63	4.99 | 4.00 (46, 54)	**1.63**
unif-k3-r4.2-v4000-c16800-S1580061366-10	2.51 | 1.98	19.21 | 13.11	1.18 | 1.01 (42, 58)	**2.13**
unif-k3-r4.2-v4000-c16800-S1588170820-15	94%	14%	11% (5, 6)	≤ 1
unif-k3-r4.2-v4000-c16800-S1946534526-16	32.55 | 26.42	99%	74.09 | 53.54 (39, 61)	0.44
unif-k3-r4.2-v4000-c16800-S251010207-06	25.84 | 19.71	319.55 | 225.40	12.70 | 9.04 (42, 58)	**2.03**
unif-k3-r4.2-v4000-c16800-S313074252-14	8.11 | 5.25	89.64 | 71.40	2.46 | 2.06 (44, 56)	**3.30**
unif-k3-r4.2-v4000-c16800-S417670629-08	29.57 | 19.12	98%	20.68 | 14.36 (39, 61)	**1.43**
unif-k3-r4.2-v4000-c16800-S440063851-05	2.40 | 1.78	17.06 | 10.64	1.24 | 1.14 (39, 61)	**1.94**
unif-k3-r4.2-v4000-c16800-S450187849-03	10%	0%	0% (0, 0)	≤ 1
unif-k3-r4.2-v4000-c16800-S669431406-04	20.39 | 12.51	208.07 | 138.03	6.65 | 5.13 (48, 52)	**3.07**
unif-k3-r4.2-v7000-c29400-S102550125-14	42.92 | 31.46	82%	6.85 | 5.70 (28, 72)	**6.27**
unif-k3-r4.2-v7000-c29400-S1312035429-13	288.97 | 184.15	1%	246.31 | 160.57 (40, 60)	**1.17**
unif-k3-r4.2-v7000-c29400-S2051531193-03	45.65 | 27.33	56%	9.84 | 7.48 (30, 70)	**4.64**
unif-k3-r4.2-v7000-c29400-S2118640909-06	23.84 | 17.10	89%	4.03 | 3.08 (38, 62)	**5.92**
unif-k3-r4.2-v7000-c29400-S427890210-10	22.69 | 16.40	92%	4.31 | 3.60 (26, 74)	**5.26**
unif-k3-r4.2-v7000-c29400-S565127616-04	99%	10%	21.95 | 17.34 (31, 69)	**>1**
unif-k3-r4.2-v7000-c29400-S56884336-05	137.91 | 94.34	5%	87.66 | 60.91 (30, 70)	**1.57**
unif-k3-r4.2-v7000-c29400-S67600799-16	53.26 | 34.62	61%	4.81 | 4.45 (41, 59)	**11.07**
unif-k3-r4.2-v7000-c29400-S856579407-15	34.58 | 26.92	92%	6.33 | 4.69 (37, 63)	**5.46**
unif-k3-r4.2-v7000-c29400-S880121748-08	121.99 | 90.63	15%	25.12 | 17.88 (39, 61)	**4.86**
unif-k5-r21.3-v100-c2130-S1047920973-05	0.04 | 0.03	0.05 | 0.04	94% (93, 1)	≤ 1
unif-k5-r21.3-v100-c2130-S1180773190-07	1.28 | 1.05	0.80 | 0.60	93% (93, 0)	≤ 1
unif-k5-r21.3-v100-c2130-S455021619-18	0.02 | 0.02	0.05 | 0.04	0.04 | 0.04 (96, 3)	0.50
unif-k5-r21.3-v100-c2130-S744612847-12	0.06 | 0.04	0.06 | 0.06	0.08 | 0.06 (98, 2)	0.75
unif-k5-r21.3-v100-c2130-S804631280-01	0.11 | 0.09	0.10 | 0.08	0.13 | 0.11 (100, 0)	0.85
unif-k5-r21.3-v110-c2343-S1019153514-04	0.66 | 0.47	0.33 | 0.26	94% (94, 0)	≤ 1
unif-k5-r21.3-v110-c2343-S1813726766-14	0.10 | 0.08	0.11 | 0.09	99% (96, 3)	≤ 1
unif-k5-r21.3-v110-c2343-S1869272420-19	0.05 | 0.04	0.06 | 0.05	99% (95, 4)	≤ 1
unif-k5-r21.3-v110-c2343-S2044406543-15	0.17 | 0.14	0.13 | 0.10	0.20 | 0.17 (98, 2)	0.85
unif-k5-r21.3-v110-c2343-S2094099432-08	0.11 | 0.07	0.11 | 0.09	99% (93, 6)	≤ 1
unif-k5-r21.3-v120-c2556-S1191693850-19	0.14 | 0.11	0.12 | 0.08	98% (96, 2)	≤ 1
unif-k5-r21.3-v120-c2556-S1376493471-11	0.30 | 0.24	0.18 | 0.12	98% (98, 0)	≤ 1
unif-k5-r21.3-v120-c2556-S1615006153-07	0.48 | 0.32	0.40 | 0.26	0.55 | 0.37 (98, 2)	0.87
unif-k5-r21.3-v120-c2556-S340864765-13	1.92 | 1.35	1.50 | 0.82	82% (82, 0)	≤ 1
unif-k5-r21.3-v120-c2556-S429936805-03	0.06 | 0.04	0.06 | 0.06	99% (94, 5)	≤ 1
unif-k5-r21.3-v130-c2769-S1032474357-17	0.55 | 0.42	0.44 | 0.36	99% (99, 0)	≤ 1
unif-k5-r21.3-v130-c2769-S1109841921-18	0.50 | 0.40	0.39 | 0.28	99% (97, 2)	≤ 1
unif-k5-r21.3-v130-c2769-S116991008-16	0.37 | 0.27	0.29 | 0.22	94% (88, 6)	≤ 1
unif-k5-r21.3-v130-c2769-S1284937235-05	0.33 | 0.24	0.27 | 0.18	99% (97, 2)	≤ 1
unif-k5-r21.3-v130-c2769-S1513299405-10	0.38 | 0.31	0.29 | 0.22	99% (93, 6)	≤ 1
unif-k7-r89-v70-c6230-S1084572666-19	1.99 | 1.45	2.43 | 1.98	2.04 | 1.49 (100, 0)	0.98
unif-k7-r89-v70-c6230-S1106151685-15	0.64 | 0.38	1.43 | 1.26	99% (99, 0)	≤ 1
unif-k7-r89-v70-c6230-S1533440099-09	1.13 | 0.79	1.86 | 1.69	98% (98, 0)	≤ 1
unif-k7-r89-v70-c6230-S1635684145-01	0.52 | 0.38	1.28 | 1.15	0.55 | 0.40 (100, 0)	0.95
unif-k7-r89-v70-c6230-S1907907390-05	0.57 | 0.37	1.40 | 1.26	0.61 | 0.40 (100, 0)	0.93
unif-k7-r89-v75-c6675-S1299158672-14	11.33 | 8.29	10.93 | 6.86	98% (98, 0)	≤ 1
unif-k7-r89-v75-c6675-S1534329206-02	2.67 | 2.04	3.02 | 2.17	99% (99, 0)	≤ 1

Table 1. (*continued*)

Instance	gNovelty+	adaptG2-WSAT0	hybridGM (gNov,March)	Gain
unif-k7-r89-v75-c6675-S1572638390-17	9.99 | 7.29	10.98 | 8.39	10.26 | 7.53 (100, 0)	0.97
SAT 2007 Competition industrial instances				
vmpc_24	84.41 | 69.67	19.12 | 14.58	160.45 | 117.59 (100, 0)	0.53
vmpc_25	321.18 | 272.56	107.23 | 78.70	92% (92, 0)	≤ 1
vmpc_26	99%	160.83 | 116.69	69% (69, 0)	≤ 1
vmpc_27	93%	95.06 | 56.37	77% (77, 0)	≤ 1
SAT 2007 Competition crafted instances				
QG7a-gensys-brn004.sat05-3669.reshuffled-07	59%	36.31 | 31.48	86% (75, 11)	**>1**
QG7a-gensys-brn100.sat05-3765.reshuffled-07	84%	27.91 | 22.67	95% (86, 9)	**>1**
QG7a-gensys-ukn001.sat05-3841.reshuffled-07	84%	44.92 | 32.18	85% (81, 4)	**>1**
QG7a-gensys-ukn005.sat05-3845.reshuffled-07	89%	46.01 | 37.03	85% (77, 8)	≤ 1
sat-grid-pbl-0200.sat05-1339.reshuffled-07	71%	81%	62% (62, 0)	≤ 1
SATLIB industrial instances				
bw_large.c	4.31 | 2.94	3.49 | 2.72	2.30 | 1.77 (21, 79)	**1.87**
bw_large.d	21.34 | 15.30	16.11 | 10.63	30.78 | 27.91 (27, 0)	0.69
g125.17	11.00 | 7.50	2.17 | 1.53	6% (6, 0)	≤ 1
g125.18	0.13 | 0.13	0.22 | 0.22	0.16 | 0.16 (100, 0)	0.81
g250.15	0.25 | 0.25	1.07 | 1.07	0.34 | 0.35 (100, 0)	0.74
g250.29	13.92 | 10.11	8.96 | 7.96	14.81 | 13.41 (76, 0)	0.94
qg1-08	202.40 | 166.45	4.53 | 4.31	71% (70, 1)	≤ 1
qg2-08	43%	10.77 | 9.07	5% (5, 0)	≤ 1
qg5-11	2%	64%	5% (5, 0)	**>1**
qg6-09	92%	2.95 | 1.64	80% (68, 12)	≤ 1
qg7-13	0%	18%	0% (0, 0)	≤ 1
SATLIB crafted instances				
par16-1-c	4.49 | 2.96	12.20 | 8.56	70% (66, 4)	≤ 1
par16-2-c	51.64 | 40.79	101.65 | 81.98	66% (52, 14)	≤ 1
par16-3-c	18.24 | 14.20	30.18 | 22.65	99% (96, 3)	≤ 1
par16-4-c	19.77 | 11.30	24.91 | 17.13	97% (96, 1)	≤ 1
par16-5-c	19.21 | 12.32	17.20 | 13.45	98% (95, 3)	≤ 1
par32-1-c	0%	0%	0% (0, 0)	≤ 1
par32-2-c	0%	0%	0% (0, 0)	≤ 1
par32-3-c	0%	0%	0% (0, 0)	≤ 1
par32-4-c	0%	0%	0% (0, 0)	≤ 1
par32-5-c	0%	0%	0% (0, 0)	≤ 1

SAT 2007 Competition random benchmark. Since $2 + p$ formulae are usually under-constrained, *March_ks* is able to extend a SSP quite easily to a solution. When taking a closer look at the *hybridGM* column in table 1, one can see that the gain is larger when the number of solutions found by *March_ks* is larger as well. It is interesting to note, that a call of *March_ks* creates very little overhead. Furthermore, *hybridGM* dominates *gNovelty+* on large size uniform random 3-SAT formulae, because of the facts mentioned in section 4.

On smaller random 3-SAT formulae, *gNovelty+* and *hybridGM* perform equally well (in terms of real time, not gain-factor). In the case of of 5- and 7-SAT instances the created SSP rarely contains a solution so that *hybridGM* can not solve these instances faster than *gNovelty+*. The reason for this has not yet been discovered. Actually, *hybridGM* and *gNovelty+* also perform similar on 5- and 7-SAT instances apart from the little overhead produced my *March_ks*. The unsuccessful runs of *hybridGM* were only due to the little memory leak of *March_ks*, that unfortunately still exists. That is why we sometimes received signal 11.

Concerning the crafted and industrial instances, both *gNovelty+* and *hybridGM* perform quite bad, as expected. *gNovelty+* and *hybridGM* are simply not designed to solve industrial and crafted formulae. One might think, that *hybridGM* should have an advantage over *gNovelty+* because it uses *March_ks*. *March_ks* is a DPLL solver and DPLL solvers usually perform well on crafted and industrial instances compared to SLS algorithms. However, *March_ks* is a DPLL solver designed to solve random formulae. Therefore, *March_ks* is not a big help when trying to solve industrial/crafted instances.

8 Conclusions and Future Work

We have presented a novel and simple approach to create an incomplete hybrid SAT solver. This approach became manifested in the term of a Search Space Partition (SSP). We defined this new term, explained how such SSPs are constructed and how they are used. We implemented our novel approach in the hybrid SAT solver *hybridGM*, utilizing *gNovelty+* as the SLS component and *March_ks* as the DPLL component.

We performed an empirical study to test our new solver against *gNovelty+* (and *adaptg2wsat0* as a reference solver). This study revealed, that *hybridGM* outperforms *gNovelty+* on $2 + p$ and large size uniform random 3-SAT formulae, without experiencing serious losses in other formula categories. We also showed, that the DPLL component *March_ks* of *hybridGM* means no further advantage on (the tested) crafted and industrial instances.

However, several findings of this study raise questions that remain unanswered. For example, on uniform random 5- and 7-SAT instances, *March_ks* almost never finds a solution. Even though we provided an intuitive explanation for this in the previous section, further research is needed to explain this behavior of *hybridGM*.

Furthermore, we believe that it would be beneficial to dynamically adapt the barrier (i.e. the required number of unsatisfied clauses of local minima we consider for the creation of SSPs) while *hybridGM* performs a search. This could be used to control the number of used minima for the construction of SSPs, and thereby the number of times *March_ks* is called in order to optimize its workload.

Acknowledgments

We would like to thank the bwGrid [3] project and especially the local coordinator of the project Christian Mosch. We would also like to thank Marijn Heule for providing us with some major bug-fixes of *March_ks*.

References

1. Cook, S.: The Complexity of Theorem-Proving Procedures. In: Proceedings of the 3rd ACM Symposium on Theory of Computing, vol. 1, pp. 151–158 (1971)
2. Crawford, J.M.: Solving satisfiability problems using a combination of systematic and local search. In: Second DIMACS Challenge, Rutgers University, NJ (1993)

3. The bwGRiD, http://www.bw-grid.de/
4. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3), 201–215 (1960)
5. Ferris, B., Fröhlich, J.: WalkSAT as an Informed Heuristic to DPLL in SAT Solving. Technical report, CSE 573: Artificial Intelligence (2004)
6. Fang, H., Ruml, W.: Complete Local Search for Propositional Satisfiability. In: Association for the Advancement of Artificial Intelligence (AAAI 2004), pp. 161–166 (2004)
7. Fang, L., Hsiao, M.: A New Hybrid Solution to Boost SAT Solver Performance. In: Design, Automation, and Test in Europe, pp. 1307–1313 (2007)
8. Gableske, O.: Towards the Development of a Hybrid SAT Solver. Diploma Thesis, University of Ulm, Germany (January 2009), http://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Mitarbeiter/gableske/DA.pdf
9. Habet, D., Li, C.M., Devendeville, L., Vasquez, M.: A Hybrid Approach for SAT. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 172–184. Springer, Heidelberg (2002)
10. Havens, W., Dilkina, B.: A Hybrid Schema for Systematic Local Search. In: Tawfik, A.Y., Goodwin, S.D. (eds.) Canadian AI 2004. LNCS, vol. 3060, pp. 248–260. Springer, Heidelberg (2004)
11. Heule, M., van Maaren, H.: Effective incorporation of Double Look-Ahead Procedures. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 258–271. Springer, Heidelberg (2007)
12. Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Proceedings of AAAI 2002, pp. 635–660 (2002)
13. Hutter, F., Tompkins, D.A., Hoos, H.H.: Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 233–248. Springer, Heidelberg (2002)
14. Jussien, N., Lhomme, O.: Local Search With Constraint Propagation and Conflict-Based Heuristics. In: 7th National Conference on Artificial Intelligence, pp. 169–174 (2002)
15. Letombe, F., Marques-Silva, J.: Improvements to hybrid incremental SAT algorithms. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 168–181. Springer, Heidelberg (2008)
16. Li, C.M., Huang, W.Q.: Diversification and determinism in local search for satisfiability. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 158–172. Springer, Heidelberg (2005)
17. Mazure, B., Sais, L., Gregoire, E.: Boosting complete techniques thanks to local search methods. Annals of Mathematics and Artificial Intelligence 22, 319–331 (1998)
18. Pham, D.N., Thornton, J.R., Gretton, C., Sattar, A.: Advances in Local Search for Satisfiability. In: Australian Conference on Artificial Intelligence 2007, pp. 213–222 (2007)
19. Selman, B., Levesque, H., Mitchell, D.: A New Method for Solving Hard Satisfiability Problems. In: Proceedings of the Tenth National Conference on Artificial Intelligence, pp. 440–446. AAAI Press, Menlo Park (1992)
20. The SAT Competition Homepage, http://www.satcompetition.org
21. The SATLIB Benchmark problems, http://www.cs.ubc.ca/~hoos/SATLIB
22. Wu, Q., Hsiao, M.S.: A New Simulation-Based Property Checking Algorithm Based on Partitioned Alternative Search Space Traversal. IEEE Transactions on Computers 55(11) (2006)
23. Zhang, W.: Configuration landscape analysis and backbone local search. Part I: Satisfiability and maximum satisfiability. Artificial Intelligence 158, 1–26 (2004)

Building a Hybrid SAT Solver via Conflict-Driven, Look-Ahead and XOR Reasoning Techniques

Jingchao Chen

School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P.R. China
chen-jc@dhu.edu.cn

Abstract. The paper develops a new hybrid SAT solver, called MoRsat. The framework of the solver is based on a look-ahead technique, and its core is a conflict-driven search. A look-ahead technique is used to split the original problem to sub-problems, each of them is either solved or aborted by a conflict-driven DPLL. Aborted sub-problems are solved recursively. We present new properties of XOR clauses, which are used to incorporate XOR reasoning into our conflict-driven DPLL. Compared with the latest versions of Rsat and March, the Gold Medal winners in the industrial and handmade SAT category of the SAT 2007 competition, MoRsat achieves remarkable improvements. Moreover, MoRsat can solve some industrial instances that were not solved in the SAT 2007 competition.

Keywords: Boolean satisfiability (SAT), Conflict-driven, Look-ahead, XOR reasoning, Hybrid solving technique, search pruning technique.

1 Introduction

A Boolean satisfiability (SAT) problem deals with the following question: For a given propositional formula, does there exist an assignment of truth values (1 or 0) to its variables for which each clause in that formula evaluates to be true. The problem is represented in Conjunctive Normal Form (CNF), in which a formula is a conjunction of clauses, each clause being a disjunction of Boolean literals, where each literal is either a variable or the negation of a variable. Many problems in the application domain such as computer aided design, artificial intelligence, cryptanalysis, planning, equivalence checking, model checking, test pattern generation etc., can be formulated as SAT problems. SAT is the first problem showed to be NP-complete [3]. Since this problem was posed, many researchers have been involved in studying it. So far, numerous SAT solvers have been developed. However, many SAT instances still remain unsolvable.

Modern SAT solvers can be divided into three categories: Conflict-driven (Rsat [1], Minisat [13], vallst, zChaff [5], eSAT), look-ahead (kcnfs, March, OKsolver) and local search (adaptnovelty, WalkSat, unitwalk, SAPS, PAWS, DLM). Each has its own strong points. Conflict-driven solvers are superior on industrial

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 298–311, 2009.

instances, look-ahead solvers are strong on unsatisfiable random and crafted instances, while local search solvers are very strong on large satisfiable random instances. We study the challenging subject how to integrate various advantages of these solvers to build a more efficient SAT solver.

Although both the conflict-driven and look-ahead type of solvers are based on a DPLL (Davis-Putnam-Logemann-Loveland) procedure, which is a complete, systematic depth-first search process, the features and data structure of the two categories are very different. The techniques included in conflict-driven solvers have watched literals scheme [5,6], learning mechanisms, restart strategies, constraint database management (clause deletion mechanisms) and branching heuristic (variable selection heuristic), while the techniques included in look-ahead solvers have adaptive double-look [16], pre-selection heuristics for partial look-ahead, dynamic addition of binary resolvents and time stamp scheme [2,15]. Notice, learning mechanisms in conflict-driven solvers are not compatible with addition of binary resolvents in look-ahead solvers. Therefore, up to now, no solver combines the conflict-driven and look-ahead techniques.

The goal of the paper is to provide a framework to boost the performance of the SAT solver via a combination of the conflict-driven and look-ahead technique. The basic idea is to use a look-ahead technique to decompose the original problem into sub-problems, each of them is either solved or aborted by a conflict-driven DPLL. Aborted sub-problems are solved in a recursive way. The look-ahead technique used in this paper is similar to March [2], but simplified. We removed the double-look and simplified the branch decision heuristic in March. The conflict-driven technique used in this paper is similar to Rsat [1], but improved. We improved the restart strategies and constraint database management. Due to the problem with data structure, existing conflict-driven solvers are difficult to perform XOR reasoning so that they cannot solve some well-structured instances. In order to tackle this problem, we present new properties of XOR clauses, by which we incorporate XOR reasoning and extend the existing watched literals scheme so that the conflict-driven DPLL can also analyze efficiently XOR clauses. The resulting hybrid solver is called MoRsat. The performance of MoRsat is significantly better than Rsat and March, which won Gold Medals in the industrial and handmade SAT category at the SAT 2007 competition [17], respectively. On the handmade category, MoRsat can outperform March. On the industrial category, MoRsat is faster than Rsat in many instances, and can solve some industrial instances that were not solved in the SAT 2007 competition. On the random category, MoRsat is slower than March. However, the number of random instances solved by MoRsat is almost the same as that solved by March within 5000 seconds.

2 Embedding XOR Reasoning into Conflict-Driven DPLL

In general, conflict-driven solvers, e.g. Rsat [1], MiniSat [13], have difficulties to solve structured instances such as the 32-bit parity problem. However, look-ahead solvers such as March [2] can solve them easily. The reason is that the

parity problem contains many so called XOR (exclusive-or, which is called equivalence in [9,15]) clauses, and the March solver does a considerable amount of reasoning during the pre-processing phase, and a minimal amount of XOR reasoning during the solving phase. To enhance the performance, we decided to modify the conflict-driven solver and its data structure in such a way that it can apply also XOR reasoning. During the pre-processing phase, without any modification, the conflict-driven solver can perform XOR reasoning. However, during the solving phase, because of the watched-literals scheme and conflict resolving (clause learning), we need overcome some obstacles, which will be discussed below.

For the sake of the discussion, we formalize some concepts by notations. A CNF formula $\mathcal{F}$ can be formulated as

$$\mathcal{F} = \bigwedge_{i=1}^{m} C_i$$

where C_i is a clause, which can be denoted by the following notations:

$$C_i = \bigvee_{j=1}^{k} x_j$$

where x_j is a literal, which is either a variable or the negation of a variable. Let $\oplus$ stand for a XOR operation, i.e. equivalently modulo 2 arithmetic. A XOR clause is defined as

$$x_1 \oplus x_2 \oplus \cdots \oplus x_n = 1$$

where $x_i (i = 1, 2, \ldots n)$ is a literal. This is equivalent to that the sum of $x_1, x_2, \ldots, x_n$ is odd. For any literal x, we have

$$\neg x = x \oplus 1$$

Hereafter, we will use this formula to denote $\neg$ (negation) operation. Similarly, we will use $x = x \oplus 0$ to denote x itself. By observing the relation between XOR clauses and CNF clauses, we obtain the following lemma.

Lemma 1. *$x_1 \oplus x_2 \oplus \cdots \oplus x_n = 1$ iff the CNF formula $\bigwedge_{y_1 \oplus y_2 \oplus \cdots \oplus y_n = 0} \bigvee_{i=1}^{n} (x_i \oplus y_i)$ is true, where x_i is a literal, and y_i is a Boolean constant in $\{0, 1\}$.*

Proof. It is straightforward. The XOR formula $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 1$ means that the sum of $x_1, x_2, \ldots, x_n$ is odd. In each clause C: $(x_1 \oplus y_1) \vee (x_2 \oplus y_2) \vee \cdots \vee (x_n \oplus y_n)$, the number of $\neg x_i = x_i \oplus y_i$ is even, since $y_1 \oplus y_2 \oplus \cdots \oplus y_n = 0$. In other words, when the XOR formula holds, for each clause C, there exists at least one x_j such that $(x_j = 1 \wedge y_j = 0)$ or $(x_j = 0 \wedge y_j = 1)$. Conversely, the CNF formula holds, the number of $x_j = 1$ must be odd. Otherwise, setting $y_j = 1$ when $x_j = 1$ yields that a clause is false, which is contradiction. □

Clearly, the following lemma holds.

Lemma 2. *Suppose $y_1, y_2, \ldots, y_n$ are Boolean constants in $\{0, 1\}$. If $y_1 \oplus y_2 \oplus \cdots \oplus y_n = 0$, then $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 1$ is equivalent to $(x_1 \oplus y_1) \oplus (x_2 \oplus y_2) \oplus \cdots \oplus (x_n \oplus y_n) = 1$, where $x_i \oplus y_i$ corresponds to $\neg x_i$ when $y_i = 1$, and x_i otherwise.*

Unlike the March solver [2], which uses a multiplicative representation to derive an XOR constraint, our solver uses an additive representation to derive an

XOR constraint. Based on Lemma 1, the XOR constraints can be recognized syntactically from the given CNF formula $\mathcal{F}$. This extracting process is very simple. It can be done by identifying whether for all $y_1 \oplus y_2 \oplus \cdots \oplus y_n = 0$, $(x_1 \oplus y_1) \vee (x_2 \oplus y_2) \vee \cdots \vee (x_n \oplus y_n)$ is in $\mathcal{F}$, where $x_i \oplus y_i (i = 1, \ldots, n)$ corresponds to $\neg x_i$ when $y_i = 1$, and x_i otherwise. If it is true, $\mathcal{F}$ contains the XOR constraint $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 1$. Consider the example formula $\mathcal{F}$:

$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$.

We can derive XOR constraint $x_1 \oplus x_2 \oplus x_3 = 1$, since $y_1 \oplus y_2 \oplus y_3 = 0$ implies in total four cases $\langle y_1, y_2, y_3 \rangle = \langle 0, 0, 0 \rangle, \langle 1, 1, 0 \rangle, \langle 1, 0, 1 \rangle$, or $\langle 0, 1, 1 \rangle$, and in each case, $(x_1 \oplus y_1) \vee (x_2 \oplus y_2) \vee (x_3 \oplus y_3) \in \mathcal{F}$. In the final implementation, XOR constraints are extracted by sorting CNF clauses, and counting the number of negated literals in each clause.

Some instances such as the parity problem contain a large amount of XOR constraints. Only the extraction operation of XOR constraints is not sufficient to solve them. Like the March solver [2], we perform additional optimization operations on detected XOR constraints. These optimization operations include the separation of independent and dependent variables, XOR constraint substitution, the elimination of tautological XOR constraints, and the shortening of XOR constraints. For the implementation details of the optimizations, we refer the reader to [18].

We can store XOR clauses in the same way as CNF clauses. However, this makes an impact on conflict-driven solvers. The main impact is the watched-literals scheme and keeping track of implication reasons. In contrast to CNF clauses, XOR constraints requires more than two watch pointers. In order to ensure that any truth value change in the watched literals is propagated in XOR clauses, in addition to two watched literals l_1 and l_2, we maintain their negations $\neg l_1$ and $\neg l_2$ for each XOR clause. So we use 2×2 watched literals $l_1, l_2, \neg l_1$ and $\neg l_2$ to watch each XOR cluase, instead of 2 literals.

Conflict-driven solvers use *clause learning*, which plays a critical role in improving the efficiency of modern SAT solvers. Its goal is to cache "causes of conflict", which are derived by resolving the current conflict. To do this, we need maintain the implication reasons, which are described by the implied literal and the clause associated with an implication. To record an implication reason, when a CNF clause becomes *implied*, that clause will be updated into the following form: the implied literal followed by the assigned literals with value 0. This technique cannot directly be applied to XOR clauses, since when an XOR clause becomes *implied*, the assigned literals in that clause are not necessarily of value 0. However, we can also maintain the implication reasons of XOR clauses by making a slight modification: negating the implied literal with value 0 and the assigned literals with value 1. In detail, first, we swap the implied literal to the first position and place the other literals after it. Second, change into its negation when it is either the implied literal with value 0 or an assigned literal with value 1, and do absolutely nothing otherwise. From Lemma 2, it is easy to verify that the original XOR clause and the XOR clause newly created by negation operations are equivalent. Given an XOR clause $C_1 : x_1 \oplus x_2 \oplus \cdots \oplus x_n = 1$, where x_1

is an implied literal, while the others are assigned. We set y_1 to 1 when the value of x_1 is 0, and 0 otherwise, set $y_i (i = 2, 3, \ldots, n)$ to 1 when the value of x_i is 1, and 0 otherwise. Clearly, $y_1 \oplus y_2 \oplus \cdots \oplus y_n = 0$. The XOR clause newly created by negation operations is viewed as $C_2 : (x_1 \oplus y_1) \oplus (x_2 \oplus y_2) \oplus \cdots \oplus (x_n \oplus y_n) = 1$. From Lemma 2, the clauses C_1 and C_2 are equivalent. Furthermore, it can be guaranteed that the implied literal $x_1 \oplus y_1$ (which corresponds to $\neg x_1$ when $y_1 = 1$, and x_1 otherwise) is forced to 1, and all the other literals $(x_i \oplus y_i)$ are assigned to value 0. Therefore, conflict-driven SAT solvers can also use the data structure of CNF clauses to perform clause learning on XOR clauses.

3 A Variant of Conflict-Driven DPLL

Our SAT solver MoRsat is built on the top of Rsat [14]. We will solve each subproblem by an improved Rsat. The core of Rsat is a conflict-driven DPLL solver. To avoid spending too much time in branches in which finding an answer is very uncertain, we escape from the morass before the search procedure terminates by setting a parameter *cutoff*, which is similar to a timeout parameter. The variant of conflict-driven DPLL used in our solver may be described in the following pseudo-code:

Algorithm 1. ConflictDrivenDPLL($\mathcal{F}$, *cutoff*)
```
#conflicts := 0
while true do
    if no unassigned vars then return satisfiable
    lit := GetBranchLiteral()
    F := BCP(F ∪ {lit})
    while F contain empty clause do
        blevel := AnalyzeConflict()
        if blevel = 0 then return unsatisfiable
        #conflicts := #conflicts+1
        if #conflicts > cutoff then return unknown
        F := Backtrack(blevel )
        F := BCP(F)
    end while
end while
```

Except that parameter *cutoff* is used to control the search space, Algorithm 1 is the same as the usual conflict-driven DPLL. Procedure GetBranchLiteral() is to select a variable unassigned so far, and assign it a value as a return. Here, various variable selection strategies can be applied. One used in MoRsat is VSIDS (Variable State Independent Decaying Sum) [5].

Procedure BCP($\mathcal{F}$) performs Boolean Constraint Propagation on formula $\mathcal{F}$. In detail, if a clause consists only of literals assigned to value 0 (false) and one unassigned literal, this procedure fixes that unassigned literal to the value 1 (true) to satisfy that clause. This variable assignment is referred to as an implication. The clause associated with an implication is said to be a unit clause.

Procedure BCP carries out repeatedly the identification of unit clauses and the creation of the associated implications until either no more implications are found or a conflict (empty clause) is produced.

Procedure AnalyzeConflict() is to determine the depth of the search tree to which the algorithm should backtrack. This is not a chronological backtracking. The concrete backtracking depth depends on conflict resolving algorithms [1,5,13]. When the backtracking level goes beyond the root of the tree, i.e. $blevel = 0$, the procedure terminates and indicates that the problem is unsatisfiable. The operation of Backtrack($blevel$) is to invalidate any implications with decision levels $\geq blevel$ to which we backtracked, and flip simply the value of the decision assignment at that backtracking level. Then, we continue to carry out BCP with the $\mathcal{F}$ in the new state. Another role of AnalyzeConflict() is to add one or more conflict clauses describing the resolved conflict to the clause learning database. This is to avoid the repeated search.

The BCP operation is time-consuming. It is reported that about 90% of the solver's run time is spent in the BCP process [5]. Therefore, Moskewicz et al. [5] proposed a BCP optimizing strategy, called the watched-literals scheme, which has been used by most SAT solvers. This can be traced back to earlier lazy data structures introduced by Zhang [6] in the solver SATO. This scheme is based on the fact that any active (i.e., not yet satisfied) clause has at least two unassigned literals. The basic idea of this scheme is to maintain two special literals to watch for each active clause. Without the watch scheme, for a clause with n literals, we need to visit it $n-1$ times in order to set one of its literals to true. With the watch scheme, we need to visit it only when one of its two watched litreals is assigned to false. To achieve this goal, when a clause is visited, we need to perform the following operation: if a clause is not unit, we choose one non-watched literal to replace the one just assigned to 0 (false). This is guaranteed that each clause has still two watched literals. If a clause is unit, we set the other watched literal to true. The main benefit of the watch scheme is to save the costs of visiting long clauses and reassigning a variable.

4 A Hybrid SAT Solver

Look-ahead solvers, such as March [2], can easily solve some crafted CNF instances, while conflict-driven solvers cannot. In order to enable our solver to solve them like look-ahead style solvers, we decided to use a look-ahead technique as a basic framework to decompose the original problem into sub-problems, each of them corresponds to a branch of the search space tree. Before entering a branch, we try to solve it by a conflict-driven DPLL. This search space is limited by the number of conflicts collected so far. In general, the value of *cutoff* is restricted to be very small, but becomes sharply large at some search-tree depth. If the try fails, we continue to search each branch along the look-ahead framework. Compared with the usual look-ahead framework, our search framework adds only a call to the conflict-driven DPLL given above. Algorithm 2 shows the pseudo-code of our SAT solver.

When calling ConflictDrivenDPLL, we use function cutoff(*level*) to compute the value of *cutoff*, which depends on the current level and the nature of SAT problems. In our implementation, in most cases, cutoff(*level*) is set to

$$\text{cutoff}(level) = \begin{cases} 300000 & level = 0 \\ 5000 & level < 8 \\ 130000 & level \geq 8 \end{cases}$$

Up to now, we have no the theoretical evidence that what the optimal value of cutoff(*level*) is. Actually, the value of *cutoff* is adjustable. Based on our observation and intuition, if the number of assigned variables is less than the number of variables/4, even if $level \geq 8$, it is better to set *cutoff* to 5000. For large SAT instances, if the maximal $H(x_i)$ value (which is defined in Algorithm 3) is very small, we tend to set the initial *cutoff* (i.e. the case of $level = 0$) to a larger value, say 500000.

Algorithm 2. SATsolver($\mathcal{F}, level$)
if $\mathcal{F} = \emptyset$ **then return satisfiable**
if empty clause $\in \mathcal{F}$ **then return unsatisfiable**
$\langle l_{branch}, Ret \rangle$:= LookAhead($\mathcal{F}$)
if $Ret =$ **unsatisfiable then return unsatisfiable**
Ret := ConflictDrivenDPLL($\mathcal{F}$,cutoff($level$))
if $Ret \neq$ **unknown then return** Ret //either SAT or UNSAT
if SATsolver($\mathcal{F}(l_{branch=1}), level + 1$) = **satisfiable then return satisfiable**
return SATsolver($\mathcal{F}(l_{branch} = 0), level + 1$)

Algorithm 3. LookAhead($\mathcal{F}$)
for each variable x_i in $\mathcal{P}$ **do**
$\mathcal{F}' := \mathcal{F}(x_i = 0)$
$\mathcal{F}'' := \mathcal{F}(x_i = 1)$
if empty clause $\in \mathcal{F}'$ and empty clause $\in \mathcal{F}''$ **then return unsatisfiable**
if empty clause $\in \mathcal{F}'$ **then** $\mathcal{F} := \mathcal{F}''$
else if empty clause $\in \mathcal{F}''$ **then** $\mathcal{F} := \mathcal{F}'$
else $H(x_i)$:= ACE($x_i, \mathcal{F}, \mathcal{F}'$)×ACE($\neg x_i, \mathcal{F}, \mathcal{F}''$)
end for
$v_{branch} := x_i$ with greatest $H(x_i)$
return GetDirection(v_{branch})

In the procedure shown in Algorithm 3, $\mathcal{P}$ denotes the pre-selected set, on which a look-ahead procedure is performed. For small SAT instances, say $\#var < 10000$, $\mathcal{P}$ is actually the set of all the unassigned variables. However, for large SAT instances, performing a look-ahead procedure on all unassigned variables is very costly. Therefore, in this case, we restrict $\mathcal{P}$ to a subset of the unassigned variables. The selection criterion of the subset depends on the variable ranking. Variables with the highest ranking are usually selected. The ranking of a variable

is determined by the number of its occurrences in CNF and XOR clauses. The higher the number of the occurrences, the higher the ranking.

The notation $\mathcal{F}(x_i{=}0)$ denotes the resulting formula after assigning x_i to false and performing iterative unit propagation. $\mathcal{F}(x_i{=}1)$ is similar. When the two look-ahead procedures both result in a conflict (the empty clause is generated), Algorithm 3 will terminate and indicate that $\mathcal{F}$ is unsatisfiable.

We rank variables by a simple heuristics function $H(x_i)$, which is defined as the product of two ACE (Approximation of the Combined lookahead Evaluation) functions. Our ACE is simpler than that used in March [2,18]. It is defined as

$$\mathrm{ACE}(x,\mathcal{F},\mathcal{F}') = \sum_{C_i \in \mathrm{CNF}(x,\mathcal{F})} CW(\mathrm{size}(C_i,\mathcal{F}')) + \sum_{C_i \in \mathrm{XOR}(x,\mathcal{F})} XW(\mathrm{size}(C_i,\mathcal{F}'))$$

where $\mathrm{CNF}(x,\mathcal{F})$ and $\mathrm{XOR}(x,\mathcal{F})$ refer to the set of CNF clauses and the set of XOR clauses in formula $\mathcal{F}$ in which variable x occurs, respectively, $\mathrm{size}(C_i,\mathcal{F}')$ denotes the length of the clause to which C_i is reduced after an iterative unit propagation $\mathcal{F}'$, and weight functions $CW(n)$ and $XW(n)$ are defined as

$$CW(n) = 5^{2-n}$$
$$XW(n) = 5.5 \times 0.85^n$$

where n is the length of the reduced clause. The computational overhead for the heuristics function $H(x_i)$ is very cheap.

The use of watch pointers makes it difficult to compute the length of reduced clauses. For this reason, we maintain two kinds of data structures: full pointers and watch pointers. In the full-pointers data structure, each literal is placed in the watch list. This data structure is used to compute the length of reduced clauses in the look-ahead procedure. The watch-pointers data structure is used in the conflict-driven solving procedure. In our implementation, we do not add any learned clauses to the full-pointers data structure to reduce the maintenance cost, since maintaining it dynamically is expensive. While computing the length of reduced clauses, we propagate failed literals, but we do not learn any local resolvents.

Function GetDirection is used to calculate the branch direction. Its value is determined by ACE. When $\mathrm{ACE}(\neg v,\mathcal{F},\mathcal{F}(v{=}1)) \leq \mathrm{ACE}(v,\mathcal{F},\mathcal{F}(v{=}0))$, we enter first the latter branch $\mathcal{F}(v{=}0)$, otherwise the former. Note that the branch direction obtained here is opposite to the one used in March [2].

5 Various Optimizations

When solving sub-problems, we use the conflict-driven solver shown above. In many places, our conflict-driven solver is the same as Rsat. For example, the conflict resolution scheme (clause learning scheme) and decision heuristic used in our solver are firstUIP (unique implication points) + conflict clause minimization and VSIDS (Variable State Independent Decaying Sum), which are the same as those used in Rsat 2.01 [14]. However, in some places, our solver is different from Rsat. We made a slight modification and optimization on some strategies such as restart strategies, clause learning database maintenance etc.

5.1 Restart Strategies

Like other conflict-driven SAT solvers, our solver has also a "restart" policy. Many restart policies have been proposed [7]. The restart policy employed in our solver is a combination of two restart techniques: Geometric series and Luby et al. series. The first restart policy restarts after every x conflicts. The value of x is multiplied by y after each restart. Like Minisat [13], we initialize the parameters of this policy to $x = 100$ and $y = 1.5$. The second restart policy performs restarts according to the Luby series: 1, 1, 2, 1, 1, 2, 3,..., multiplied by a unit-run constant x. In Rsat, $x = 512$, which is the same as one used in our solver. Depending on the nature of the instance to be solved, our solver adopts different restart policies. If the instance to be solved contains XOR clauses, our solver will adopt Minisat (Geometric) restart policy. Otherwise, it will adopt the Luby restart policy. This choice is based on the fact that for instances with XOR clauses, Minisat is faster than Rsat. For example, Minisat can solve two out of five ezfact64 instaces used in SAT 2007 competition, while Rsat cannot solve any of them.

5.2 Clause Learning Database Maintenance

Like the restart policy, our clause learning database management is also a combination of two strategies: Minisat [13] strategy and Rsat [1] strategy. When a problem to be solved contains XOR clauses, we use Minisat strategy to compute the upper bound of constraint database. In other cases, we use Rsat management strategy. For instances with XOR clauses, the maximal number of learned clauses is increased by a factor of 1.04 (which is 1.05 in Minisat) after each restart, while for instances without XOR clauses, it is increased by a factor of 1.38 (which is 1.5 in Rsat) after each restart. At any time, the maximal number of learned clauses is limited to 300000. This limitation improves the search efficiency, and reduces the memory usage. When solving a new sub-problem, we do not clear some still-relevant clauses added previously to the database to avoid simply repeating previous analysis. This is similar to a restart process. Like other conflict-driven solvers, our solver supports also the deletion of added conflict (learned) clauses to avoid a memory explosion. Furthermore, our deletion strategy is completely the same as Rsat 2.01 [14].

6 Empirical Evaluation

To measure the efficiency of our solver MoRsat [19], as comparison objects, we selected two of the most representative solvers. One is the conflict-driven solver Rsat, which won a Gold Medal for the industrial category at SAT 2007. The other is the look-ahead solver March, which won a Gold Medal for handmade SAT category at SAT 2007. We selected the most recent versions, Rsat 3.01 [14] and March_ks [2]. All our experiments were done under such a platform: Intel Core 2 Quad Q6600 CPU with speed of 2.40GHz and 2GB memory. The instances used in our experiments cover three categories: random, crafted and

Table 1. Runtime comparison (in seconds). (The crafted, industrial and random benchmarks are placed in up, middle, down sub-table. ">5000" shows that the instance cannot be solved in 5000 seconds.)

Instance	Answer	MoRsat	Rsat 3.01	March_ks
cnf-r4-b1-k1.1	SAT	484.45	431.02	>5000
cnf-r4-b1-k1.2	SAT	69.97	213.68	>5000
philips	UNSAT	417.47	>5000	177.76
hwb-n26-01	UNSAT	719.33	>5000	761.16
ezfact64-1 SAT02	SAT	89.69	>5000	1002.81
ezfact64-2 SAT02	SAT	98.66	>5000	1667.55
ezfact64-3 SAT07	SAT	194.84	>5000	1862.14
ezfact64-4 SAT07	SAT	136.41	>5000	3069.36
ezfact64-5 SAT07	SAT	76.72	>5000	4625.41
ezfact64-6 SAT07	SAT	342.05	>5000	1609.57
ezfact64-7 SAT02	SAT	1229.06	>5000	>5000
ezfact64-8 SAT02	SAT	528.78	>5000	>5000
ezfact64-9 SAT02	SAT	1268.09	>5000	1846.32
ezfact64-10 SAT07	SAT	869.86	>5000	>5000
par32-1	SAT	1.86	>5000	1.38
par32-2	SAT	0.97	2640.40	1.58
par32-3	SAT	0.38	>5000	0.50
par32-4	SAT	4.16	>5000	0.22
par32-5	SAT	3.95	>5000	0.77
par32-1-c	SAT	5.98	>5000	1.34
par32-2-c	SAT	2.55	>5000	2.33
par32-3-c	SAT	1.06	>5000	2.38
par32-4-c	SAT	2.19	>5000	2.21
par32-5-c	SAT	9.14	>5000	1.34
lisa21-99-a	SAT	4.66	11.83	41.87
pb-s-40-4	SAT	195.22	305.8	402.83
dated-5-15-u	UNSAT	405.13	714.32	Out of Memory
dated-5-17-u	UNSAT	571.80	762.66	Out of Memory
dated-10-11-u	UNSAT	3600.34	3907.44	Out of Memory
dated-10-13-u	UNSAT	1711.20	1968.46	>10000
mizh-md5-47-4	SAT	75.08	1975.33	>10000
mizh-md5-47-5	SAT	67.86	1238.24	>10000
AproVE07-01	UNSAT	7098.52	>10000	>10000
AproVE07-27	UNSAT	1881.64	4162.75	>10000
9vliw-m-9stages-iq3-C1-bug1	SAT	123.45	243.73	>10000
9dlx-vliw-at-iq2	UNSAT	1278.59	437.41	>10000
unif2p-p0.7-v3500-c9345 S1286605994	SAT	142.59	622.39	26.63
unif2p-p0.7-v3500-c9345 S1377128774-12	UNSAT	54.38	35.21	17.06
unif2p-p0.7-v4500-c12015 S1311582577-17	UNSAT	2737.67	>5000	580.42

industrial, analogue to the SAT competition. Most of the instances are from SAT 2007 competition. Table 1 shows the experimental results of the instances we tested[1]. In our experiments, the timeout for each solver was 10000 seconds for each industrial instance, and 5000 seconds for each crafted or random instance. All the running times for MoRsat and March include various preprocessing times such as XOR detection. In our test cases, the preprocessing time for each instance was very short and never exceeded 10 seconds.

For the crafted category, we selected the following benchmarks:

- DES instances, cnf-r4-b1-k1.1 and cnf-r4-b1-k1.2, contributed by Massacci (1999) [9]. These are encoding of cryptographic key search and contain XOR clauses from 4 rounds;
- the philips family submitted by Heule to SAT 2004 [8], which arises from an encoding of a multiplier circuit provided by Philips;
- the hwb family used in SAT 2007. It consists of equivalence checking problems that arise in combining two circuits computed by the hidden weighted bit function - provided by Stanion [10];
- the ezfact64 family used in SAT 2002 and SAT 2007. This family is encoding of factorization problems contributed by Pehoushek [11].
- the parity32 family contributed by Crawford et al. which arises from the SAT-encoding of minimal disagreement parity problems [4];
- the lisa21-99-a contributed by Aloul [11];
- the pb-s-40-4 contributed by Pyhala Braun to SAT 2002 [11]. This is another encoding of factoring problems.

On this category, Rsat was not competitive at all. As a whole, March was also significantly slower than MoRsat. As shown in Table 1, for ezfact64, MoRsat solved easily all 10 of the benchmarks, while Rsat could not solve any instance, and March aborted three. For par32, Rsat could complete only one out of 10 benchmarks. MoRsat again completed all 10 within 10 seconds, and achieved almost the same performance as March. For most of the other crafted instances, MoRsat was faster than both Rsat and March.

For the industrial category, all the benchmarks used in our experiments come from SAT 2007 [17]. They consist of the following instances:

- the dated family contributed by anbulagan;
- the mizh-md5 family from MD5 cryptanalysis contributed by Mironov and Zhang;
- the AproVE07 family;
- the vliw-sat.4.0 and vliw-unsat.2.0 family from pipelined machine verification[12].

On this category, March was not competitive at all. It was either out of memory or timed out on this set. Except for one (9dlx-vliw-at-iq2) of the benchmarks, MoRsat was faster than Rsat. Particularly for mizh-md5, MoRsat obtained one

[1] MoRsat has been submitted to SAT 2009 competition. Therefore, for additional results we refer to the SAT 2009 competition results.

to two orders of magnitude performance improvement in comparison with Rsat. To our best knowledge, up to now, no solver can solve AProVE07-01 in a reasonable time. However, MoRsat solved it in 7098 seconds.

Figure 1 presents a log-log scatter plot comparing the runtimes of MoRsat and Rsat on some industrial instances, which are from SAT 2007, and cover all sub-categories except for the IBM benchmark. Points lying above the diagonal denote instances in which MoRsat outperforms Rsat. As can be seen, for more than half of instances, MoRsat is faster than Rsat.

For the random category, only three instances were chosen from SAT 2007 [17], as shown in the last three rows of Table 1. Because the empirical results on other random instances were similar, we omitted them. On this category, MoRsat outperformed Rsat, but was slower than March. Even so, the number of instances solved by MoRsat was almost the same as the number of instances solved by March within 5000 seconds.

In the tested instances, the maximal number of levels required by MoRsat to find a solution ranges from 0 to 25. On par32 and mizh-md5 benchmarks, the maximal number of levels is 0. On the ezfact64 benchmark, it is 10. On some DES instances, it is 25. On the random benchmark, it is about 16. If the maximal number of levels is too large, say 50, it is difficult to find a solution.

For different instances, the reason why the performance is improved varies. The improvement on some instances is only due to the XOR handling, e.g. par32, mizh-md5 etc., while the improvement on other some instances is due to the hybrid technique of look ahead and conflict driven, and the XOR handling, e.g. ezfact64, hwb-n26 etc. The reason why we omit comparison with Minisat is based on two points: 1) Except for instances with XOR constraints, in most cases,

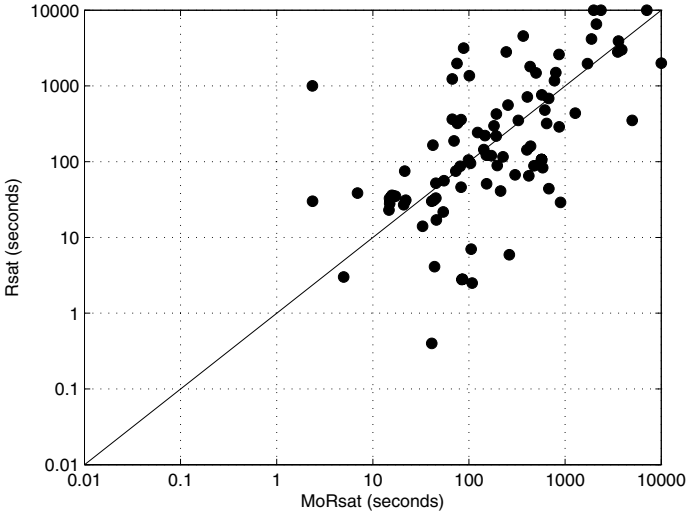


Fig. 1. Comparing the runtimes of MoRsat and Rsat on some industrial instances from SAT 2007

Minisat is not better than Rsat; 2) Our solver inherits many features from Rsat. In many cases, the performance of our solver is consistent with that of Rsat.

7 Conclusions

The conflict-driven technique seems to pay attention to the local nature, while the look-ahead technique seems to pay attention to the global nature. We combined them to build a new SAT solver. The proposed solver has a significant speed-up. The improvement is mainly obtained by the structure of the framework or meta-heuristics. The search framework given in this paper is not only applicable to the conflict-driven and look-ahead technique, but also to other techniques. On the other hand, efficiency of solvers based on the framework relies heavily on the state of the art of the basic solving technique. Using a more efficient conflict-driven and look-ahead technique, without a doubt, a more efficient solver will be obtained. However, it is possible that no matter how much the existing techniques are improveed, some SAT problems remain still unsolvable. Therefore, developing a new search framework, reasoning mechanism and analysis technique is necessary. This is our future research project.

References

1. Pipatssrisawat, K., Darwiche, A.: Rsat 2.0: SAT solver description. In: SAT 2007 competition (2007)
2. March_ks SAT solver, the version of SAT 2007 competition, http://www.st.ewi.tudelft.nl/sat/download.php
3. Cook, S.A.: The Complexity of Theorem Proving Procedures. In: 3rd ACM Symp. on Theory of Computing, pp. 151–158 (1971)
4. Crawford, J.M., Kearns, M.J., Schapire, R.E.: The Minimal Disagreement Parity Problem as a Hard Satisfiability Problem. Draft version (1995)
5. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L.T., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Design Automation Conference, DAC (2001)
6. Zhang, H.: SATO: An efficient propositional prover. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 272–275. Springer, Heidelberg (1997)
7. Ryvchin, V., Strichman, O.: Local Restarts. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 271–276. Springer, Heidelberg (2008)
8. Le Berre, D., Simon, L.: Fifty-five solvers in vancouver: The SAT 2004 competition. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 321–344. Springer, Heidelberg (2005)
9. Li, C.M.: Integrating Equivalency Reasoning into Davis-Putnam Procedure. In: AAAI-2000, Austin, Texas (2000)
10. Le Berre, D., Simon, L.: The essentials of the SAT 2003 competition. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 452–467. Springer, Heidelberg (2004)
11. Simon, L., Le Berre, D., Hirsch, E.: The SAT 2002 competition. Annals of Mathematics and Artificial Intelligence (AMAI) 43, 343–378 (2005)
12. Velev, M., Bryant, R.: Effective use of Boolean Satisfiability Procedures in the Formal Verification Superscalar and VLIW Microprocessors. In: Design Automation Conferece, DAC (2001)

13. Eén, N., Sörénsson, N.: Minisat v2.0 (beta) solvers description, SAT-race (2006), http://minisat.se
14. Rsat SAT solver homepage, http://reasoning.cs.ucla.edu/rsat/
15. Heule, M., Van Maaren, H.: Aligning CNF- and Equivalence-Reasoning. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 145–156. Springer, Heidelberg (2005)
16. Heule, M., Van Maaren, H.: Effective Incorporation of double look-ahead procedures. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 258–271. Springer, Heidelberg (2007)
17. SAT 2007 Competition Homepage, http://www.satcompetition.org/2007/
18. Heule, M.: March: towards a look-ahead SAT solver for general purposes, Master thesis (2004)
19. Chen, J.C.: The SAT solver, MoRsat. In: SAT 2009 Competition (2009) (submitted)

Restart Strategy Selection Using Machine Learning Techniques

Shai Haim and Toby Walsh

NICTA and School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
{shai.haim,toby.walsh}@nicta.com.au

Abstract. Restart strategies are an important factor in the performance of conflict-driven Davis Putnam style SAT solvers. Selecting a good restart strategy for a problem instance can enhance the performance of a solver. Inspired by recent success applying machine learning techniques to predict the runtime of SAT solvers, we present a method which uses machine learning to boost solver performance through a smart selection of the restart strategy. Based on easy to compute features, we train both a satisfiability classifier and runtime models. We use these models to choose between restart strategies. We present experimental results comparing this technique with the most commonly used restart strategies. Our results demonstrate that machine learning is effective in improving solver performance.

1 Introduction

Restarts have been shown to boost the performance of backtracking SAT solvers (see for example [11],[24]). A restart strategy (t_1,t_2,t_3,...) is a sequence of restart lengths that the solver follows in the course of its execution. The solver first performs t_1 steps (in case of SAT solvers a step is usually a conflict). If a solution is not found, the solver abandons its current partial assignment and starts over. The second time it runs for t_2 steps, and so on. Luby, Sinclair and Zuckerman [16] show that for each instance there exists t^*, an optimal restart length that leads to the optimal restart strategy (t^*,t^*,t^*,...). In order to calculate t^*, one needs to have full knowledge of the runtime distribution (RTD) of the instance, a condition which is rarely met in practical cases.

Since the RTD is not known, solvers commonly use "Universal Restart Strategies". These strategies do not assume prior knowledge of the RTD and they attempt to perform well on any given instance. Huang [11] shows that when applied with Conflict Driven Clause Learning solvers (CDCL), none of the commonly used universal strategies dominates all others on all benchmark families. He also demonstrates the great influence on the runtime of different restart strategies, when all its other parameters are fixed.

In this paper we show that the recent success in applying machine learning techniques to estimate solvers' runtimes can be harnessed to improve solvers'

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 312–325, 2009.

performance. We start by discussing the different universal strategies and recent machine learning success in Sect. 2. In Sect. 3 we present *LMPick*, a restart strategy portfolio based solver. Experimental results are presented and analyzed in Sect. 4. We conclude and suggest optional future study in Sect. 5.

2 Background

Competitive DPLL solvers typically use restarts. Most use "universal" strategies, while some use "dynamic" restart schemes, that induce or delay restarts (such as the ones presented in [1] and [22]).

Currently, the most commonly used universal strategies fall into one of the following categories:

- *Fixed strategy* - ([9]). In this strategy a restart takes place every constant number of conflicts. While some solvers allow for a very short interval between restarts, others allow for longer periods, but generally fixed strategies lead to a frequent restart pattern. Examples of its use can be found in Berk-Min [8] (where the fixed restart size is 550 conflicts) and Seige [21] (fixed size is 16000 conflicts).
- *Geometric strategy* - ([23]). In this strategy the size of restarts grows geometrically. This strategy is defined using an initial restart size and a geometric factor. Wu and van Beek [24] show that the expected runtime of this strategy can be unbounded worse than the optimal fixed strategy in the worst case. They also present several conditions which, if met, guarantee that the geometric strategy would yield a performance improvement. This strategy is used by MiniSat v1.14 [5] with initial restart size of 100 conflicts and a geometric factor of 1.5.
- *Luby Strategy* - ([16]). In this strategy the length of restart i is $u \cdot t_i$ when u is a constant "unit size" and

$$t_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

  The first elements of this sequence are 1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,1,1,... Luby, Sinclair and Zuckerman [16] show that the performance of this strategy is within a logarithmic factor of the true optimal strategy, and that any universal strategy that outperforms their strategy will not do it by more than a constant factor. These results apply to pure Las Vegas algorithms, and do not immediately apply to CDCL solvers in which learnt clauses are kept across restarts. The effectiveness of the strategy in CDCL solvers appears mixed ([11],[20]) and there is still no theoretical work that that analyzes its effectiveness in such solvers. However, Luby's restart strategy is used by several competitive solvers including MiniSat2.1 and TiniSat.
- *Nested Restart strategy* - ([2]) This strategy and can be seen as a simplified version of the Luby strategy. After every iteration the restart length grows geometrically until it reaches a higher bound, at this point the restart size

is reset to the initial value and the higher bound is increased geometrically. This strategy is used by PicoSat [2] and Barcelogic[1].

Previous work shows that restart strategies perform differently on different data sets. Huang [11] compares the performance of different strategies both for benchmark families and different benchmarks. He shows that there is no one strategy that outperformed all others across all benchmark families which suggests that adapting a strategy to a benchmark family, or even a single benchmark, could lead to performance gain. This suggests that choosing the best strategy from a set of strategies could improve the overall runtime, and for some benchmark families, improves it significantly.

Machine learning was previously shown to be an effective way to predict the runtime of SAT solvers. SatZilla [25] is a portfolio based solver that uses machine learning to predict which of the solvers it uses is optimal for a given instance. SatZilla uses an hierarchical approach [26] and can use different evaluation criteria for solver performance. SatZilla utilizes runtime estimations to pick the best solver from a solver portfolio. The solvers are used as-is, and SatZilla does not have any control over their execution. SatZilla was shown to be very effective in the SAT competition of 2007. Two other Machine Learning based approaches for local search and CDCL are presented in [13] and [3] respectively.

Ruan et al. [19] suggest a way to use dynamic programming to derive dynamic restart strategies that are improved during search using data gathered in the beginning of the search. This idea corresponds with the "Observation Window" that we will discuss in the next section. There are several differences between this work and ours. One important difference is that their technique chooses a different instance from the ensemble at each restart. While our intention is to solve each of the instances in the ensemble, it seems their technique is geared towards a different goal, where the solver is given an ensemble of instances and is required to solve as many of them as possible.

Another approach for runtime estimation was presented in our previous work [10]. In that paper we introduce a Linear Model Predictor (LMP) which demonstrates that runtime estimation can also be achieved using parameters that are gathered in an online manner, while a search is taking place, as opposed to the mostly static features gathered by SatZilla. Another difference between the methods is the way training instances are used. While SatZilla uses a large number of instances that are not tightly related, LMP uses a much smaller set of problems, but these should be more homogeneous.

3 *LMPick*: A Restart-Strategy Selector

Since restart strategies are an important factor in the performance of DPLL style SAT solvers, a selection of a good restart strategy for a given instance should improve the performance of the solver for that instance. We suggest that by using supervised machine learning, it is possible to select a good restart strategy

[1] http://www.barcelogic.org/

for a given instance. We present *LMPick*, a machine learning based technique which enhances CDCL solvers' performance.

3.1 Restart Strategies Portfolio

LMPick uses a portfolio of restart strategies from which it chooses the best one for a given instance. Following [11] we recognize several restart strategies that have shown to be effective on one benchmark family or more. We chose 9 restart strategies that represent, to our understanding, a good mapping of commonly used restart strategies.

- *luby-32* - A Luby restart strategy with a "unit run" of 32 conflicts. This strategy represents a Luby restart strategy with a relatively small "unit run". This technique was shown to be effective by Huang [11].
- *luby-512* - A Luby restart strategy with a "unit run" of 512 conflicts. This strategy represents a Luby restart strategy with a larger "unit run". This is the original restart scheme used by TiniSat, the solver we used in this study.
- *Fixed-512* - A fixed restart scheme with a restart size of 512 conflicts. Similar restart schemes that are used by solvers are BerkMin's *Fixed-550* [8], and the 2004 version of zChaff, *Fixed-700* [17].
- *Fixed-4096* - A fixed balance scheme with a restart size of 4096 conflicts. We chose this restart scheme because it balances the short and long fixed schemes, and because it performed very well in our preliminary tests.
- *Fixed-16384* - A fixed balance scheme with a restart size of 16384 conflicts. A longer fixed strategy, similar to the one used by Siege [21] (*Fixed-16000*).
- *Geometric-1.1* - A geometric restart scheme with a first restart size of 32 conflicts and a geometric factor of 1.1. inspired by the one used by Hunag [11].
- *Geometric-1.5* - A geometric restart scheme with a first restart size of 100 conflicts and a geometric factor of 1.5. This is the restart scheme used in MiniSat v1.14 [5].
- *Nested-1.1* - A nested restart strategy, with an inner value of 100 conflicts, an outer value of 1000 values and a geometric factor of 1.1. This strategy is parameterized as the one used by PicoSAT [2].
- *Nested-1.5* - A nested restart strategy, with an inner value of 100 conflicts, an outer value of 1000 values and a geometric factor of 1.5. Inspired by the results presented in [22].

3.2 Supervised Machine Learning

Satsifiable and unsatisfiable instances from the same benchmark family tend to have different runtime distributions [7]. A runtime prediction model that is trained using both SAT and UNSAT instances performs worse than a homogeneous model. It is better to train a layer of two models, one trained with satisfiable instances (M_{sat}) and the other with unsatisfiable instances (M_{unsat}). Since in most cases we do not know whether a given instance is satisfiable or not we need to determine which of the models is the correct one to query for

Set I:
1. **Number of variable**
2. **Number of clauses**

Set II:
3. **Number of variable**
4. **Number of clauses**
5. **Variable to clauses ratio**
6. **Number of binary clauses**
7. **Number of ternary clauses**
8. **Number of horn clauses**
9-12. **VCG - Variable nodes degree statistics**: mean, variation coefficient, min and max
13-16. **VCG - Clause nodes degree statistics**: mean, variation coefficient, min and max
17-20. **Number of occurrences in Horn clauses**: mean, variation coefficient, min and max
21-24. **Ratio of positive and negative occurrences of variables**: mean, variation coefficient, min and max
25. **Number of assigned variable**

Set III:
26-29. **Backjump Size**: mean, variation coefficient, min and max
30-33. **Search Depth**: mean, variation coefficient, min and max
34-37. **log(WBE) value** : mean, variation coefficient, min and max

Set IV:
38. **Number of variable**
39. **Number of clauses**
40. **Variable to clauses ratio**
41. **Number of binary clauses**
42. **Number of ternary clauses**
43. **Number of horn clauses**
44-47. **VCG - Variable nodes degree statistics**: mean, variation coefficient, min and max
48-51. **VCG - Clause nodes degree statistics**: mean, variation coefficient, min and max
52-55. **Number of occurrences in Horn clauses**: mean, variation coefficient, min and max
56-59. **Ratio of positive and negative occurrences of variables**: mean, variation coefficient, min and max
60. **Number of assigned variable**

Fig. 1. A list of features used to build models

a given instance according to its probability to be satisfiable. Previous work ([26],[4]) suggests that machine learning can be successfully used for this task as well. A classifier can be trained to estimate the probability of an instance to be satisfiable. Some classification techniques perform better than others, but it seems that for most benchmark families, a classifier with 80% accuracy or more is achievable.

Using supervised machine learning, we train models offline in order to use them for predictions online. For every training example $t \in \mathcal{T}$, where $\mathcal{T}$ is the training set, we gather the feature vector $x = \{x_1, x_2, \ldots, x_n\}$ using the features presented in section 3.3. Once the raw data is gathered, we perform a feature selection. We repeatedly remove the feature with the smallest standardised coefficient until no improvement is observed based on the standard AIC (Akaike Information Criterion). We then searched and eliminate co-linear features in the chosen set. The reduced feature vector $\hat{x}$ is then used to train a classifier and several runtime prediction models. The classifier predicts the probability of an instance to be satisfiable and the runtime models predict cpu-runtime. *LMPick* trains one classifier, but two runtime models for each restart strategy $s \in S$ (where S is the set of all participating strategies) to the total of $2|S|$ models. Each training instance is used to train the satisfiability classifier, labeled with its satisfiability class, and $|S|$ runtime models, for each model it is labeled with the appropriate runtime.

As the classifier, we used a Logistic Regression technique. Any classifier that returns probabilities would be suitable. We found Logistic Regression to be a simple yet effective classifier which was also robust enough to deal with different data sets. We have considered both Sparse Multinomial Linear Regression [15]

(suggested to be effective for this task in [25]), and the classifiers suggested by Devlin and O'Sullivan in [4], but the result of all classifiers were on par when using the presented feature vector on our datasets.

For the runtime prediction models we used Ridge Linear Regression. Using ridge linear regression, we fit our coefficient vector w to create a linear predictor $f_w(\hat{x}) = w^T \hat{x}_i$. We chose ridge regression, since it is a quick and simple technique for numerical prediction, and it was shown to be effective in the Linear Model Predictor (LMP) [10]. While LMP predicts the log of number of conflicts, in this work we found that predicting cpu-runtime is more effective as a selection criterion for restart strategies. Using the number of conflicts as a selection criterion tends to bias the selection towards frequent restart strategies for large instances. This is because an instance with many variables spends more time going down the first branch to a conflict after a restart. This work is unaccounted for when conflicts are used as the cost criterion. Hence a very frequent restart strategy might be very effective in the number of conflict while much less effective in cpu-time.

3.3 Feature Vector

There are 4 different sets of features that we used in this study, all are inspired by the two previously discussed techniques - SatZilla [25] and LMP [10]. The first set include only the number of variables and the number of clauses in the original clause database. These values are the only ones that are not normalized. The second set includes variables that are gathered before the solver starts but after removing clauses that are already satisfiable, shrinking clauses with multiple appearances and propagating unit clauses in the original formula. These features are all normalized appropriately. They are inspired by SatZilla and were first suggested in [18]. The third set include statistics that are gathered during the "Observation Window", this is a period where we analyze the behavior of the solver while solving the instance. The "Observation Window" was first used in [10]. The way the observation window is used in this study will be discussed shortly. The variables in this set are the only ones which are DPLL dependent. The last set includes the same features as the second, but they are calculated at the end of the observation window. A full list of the features is presented in Fig. 1. For further explanation about these features see [18] and [10].

1. Gather information for features sets I and II.
2. Run a first restart for 100 conflicts.
3. Run a second restart for 2000 conflicts, this restart hosts the observation window, during which feature set III is gathered. At the end of the observation window, feature set IV is gathered.
4. Predict instance's probability of satisfiability using satisfiability classifier.
5. Predict runtime with each of the Models, weighing according to the probability of satisfiability.
6. Choose the strategy with the minimal weighted runtime.

Fig. 2. Steps in the operation of a restart strategy portfolio based solver. Features sets I through IV are presented in Fig. 1.

3.4 Operation of the Solver

Once all runtime models are fitted and the satisfiability classifier is trained, we can use them to improve performance for future instances. The steps that are taken by *LMPick* are presented in Fig. 2.

Since no prediction can be made before the observation window is terminated, and since we favor an early estimation, it is important that the observation window should terminate early in the search. In our preliminary testings we have noticed that the first restart tends to be very noisy, and that results are better if data is collected in the second restart onwards. We have tried several options for the observation window location and size, eventually we opted for a first restart which is very short (100 conflicts), followed by a second restart (of size 2000) which hosts the observation window. Hence the observation window is closed and all data is gathered after 2100 conflicts.

Once the feature vector x is gathered it is used with the classifier to determine the probability of the instance to be satisfiable, $P(sat|x)$. For each of the strategies, both models are queried and a best strategy $s_b(x)$ is picked using

$$s_b(x) = \arg\min_{s \in S}[P(sat|x) \cdot M_{sat,s}(x) + P(unsat|x) \cdot M_{unsat,s}(x)].$$

The restart strategy which is predicted to be the first to terminate is picked, and the solver starts following this strategy from the next restart onwards. Although restart strategies are usually followed from the beginning of the search, we do not want to lose the learned clauses from the first 2100 conflicts. Therefore, we continue the current solving process and keep the already learnt clauses. We denote the restart sequence that takes place from the first restart to termination as $LMPick_{s_b}$. It is important to note that $s_b \neq LMPick_{s_b}$.

4 Results

4.1 Experiment Settings

In this study we used TiniSat (version 0.22) [12] as the basic solver. TiniSat is a lightweight DPLL style solver that was first presented in the SAT Race of 2006. TiniSat is a modern solver that uses clause learning and a unique decision heuristic that generally favours variables from recent assignments (as in BerkMin [8]) and uses VSIDS [17] over the literals as a backup. We chose to use TiniSat since (i) it is tuned in a way that would make comparison of restart strategies more meaningful [11] and (ii) it is a compact and straightforward implementation which allows for greater ease of use. TiniSat is not equipped with a pre-processor, and we have not used any in our study. By default, TiniSat uses a Luby restart strategy with a run unit of 512 conflicts.

All our experiments were conducted on a cluster of 14 Dual Intel Xeon CPUs with EM64T (64-bit) extensions, running at 3.2GHz with 4GB of RAM under Debian GNU/Linux 4.0. By implementing a runtime cutoff of 90 minutes per instance, we managed to complete all experiments in approximately 290 CPU days.

```
1  int cbmc_function_sat(int x) {
2      int a[ARRAY_SIZE];
3      signed low=0, high=ARRAY_SIZE;
4      while(low<high) {
5          signed middle=low+((high-low)>>1);
6          if(a[middle]<x)
7              high=middle;
8          else if(a[middle+1]>x)
9              low=middle+1;
10         else
11             return middle;
12     }
13     return -1;
14 }
```

(a) *sat*

```
1  int cbmc_function_unsat(int x) {
2      int a[ARRAY_SIZE];
3      signed low=0, high=ARRAY_SIZE;
4      while(low<high) {
5          signed middle=low+((high-low)>>1);
6          if(a[middle]<x)
7              high=middle;
8          else if(a[middle]>x)
9              low=middle+1;
10         else
11             return middle;
12     }
13     return -1;
14 }
```

(b) *unsat*

Fig. 3. Code verified by CBMC to generate the *bmc* dataset. Different instances are made using different ARRAY_SIZE values and a different number of unwinding iterations.

Table 1. Summary of features of datasets. For each dataset the following details are presented: The dataset's classification (Class), the number of instances (Ins.) and its size in MB (all file are zipped). Also, we present the time (in hours) it took for all 9 restart strategies to solve these datasets. We present the mean time (Mean), the standard deviation (SD) and the minimal and maximal time. Runtime cutoff is 5400 seconds and it is the maximal runtime per instance.

Name	Class	Ins.	Size (MB)	Runtime			
				Mean	SD	Min	Max
bmc	sat	234	4,420.0	154.42	26.41	127.93	213.27
	unsat	237	1,951.7	113.82	18.59	93.48	155.04
velev	sat	72	1,866.2	32.68	3.74	24.77	37.87
	unsat	105	953.4	70.74	1.64	68.16	73.29
crypto	sat	139	2.7	140.71	11.54	122.52	164.78
	unsat	300	5.5	14.19	1.53	11.61	16.65
rand	sat	457	1.90	76.93	14.98	54.00	105.10
	unsat	601	2.3	84.82	10.82	70.87	103.56

4.2 Benchmarks

In this study we used four different distributions of SAT instances. Instances in each data set are of various difficulty. We have omitted very easy instances that are solved before the "observation window" terminates.

- *bmc:* An ensemble of software verification problems generated using CBMC[2] verifying the C functions presented in Fig. 3. These two functions are almost identical, apart for a change in line 8, which causes the *sat* script to overflow. The different instances use different array sizes and different number of unwindings. This dataset represents an ensemble of problems that are very similar and generated by the same process. We use 234 satisfiable and 237 unsatisfiable problems.

[2] http://www.cprover.org/cbmc/

- *velev:* An ensemble of hardware formal verification problems distributed by Miroslav Velev[3]. These are well studied verification hardware benchmarks. This ensemble is not as homogeneous as *bmc* because it is a union of many small benchmark families. We use 72 satisfiable and 105 unsatisfiable instances.
- *crypto:* An ensemble of problems that are generated as part of an attack on the Bivium stream cipher, presented by Eibach, Pilz and Völkel [6]. This ensemble presents some interesting characteristics. While it is generated by a non-random process, the instances are significantly smaller than common industrial instances. The satisfiable instances we use were generated with 35 guesses, the unsatisfiable ones were generated with 40 guesses. The reason for this discrepancy is that unsatisfiable instances are harder to solve in this benchmark family, and different number of guesses renders the datasets too easy or too hard. We use 139 sat and 300 unsat instances.
- *rand:* An ensemble of 457 satisfiable and 601 unsatisfiable randomly generated 3-SAT problems with 250 to 450 variables and a clause-to-var ratio of 4.1 to 5.0.

Some further data about these data sets is presented in Table 1. We would like to draw the reader's attention to the "SD" column. This column presents the standard deviation observed when the problem is solved by all 9 restart strategies. A small value indicates that all restart strategies perform on par on this data set, while a large value indicates that runtimes are scattered.

Each data set is split into training and testing sets. Instances in the training set are used to train M_{sat}, M_{unsat} and the classifier while the instances in the testing set are only used to generate the results. We used a *10-fold* cross validation technique after randomly shuffling all instances.

4.3 Restart Strategy Portfolio Performance

Table 2 demonstrates the effectiveness of *LMPick*. We present the performance of each of the 9 restart strategies on each of the data sets. We use two matrices: The number of instances solved within the cutoff time of 90 minutes, and the total time it took to solved the entire data set. Timed out instances are counted as contributing 90 minutes to the total runtime. We then present the average performance achieved by all strategies. This solver (denoted *Random-pick*) represents the expected behavior when there is no prior knowledge regarding which of the strategies is most suitable for a given instance. The last row presents the results we get using the *LMPick* process.

This table shows that for all of the data sets *LMPick* performs better than using a randomly picked strategy, both for problems solved and for total runtime.

[3] http://www.miroslav-velev.com/sat_benchmarks.html. We use the following benchmark families: vliw_sat_2.0, vliw_sat_2.1, vliw_sat_4.0, vliw_unsat_2.0, vliw_unsat_3.0, vliw_unsat_4.0, pipe_sat_1.0, pipe_sat_1.1, pipe_unsat_1.0, pipe_unsat_1.1, liveness_sat_1.0, liveness_unsat_1.0, liveness_unsat_2.0, dlx_iq_unsat_1.0, dlx_iq_unsat_2.0, engine_unsat_1.0, fvp_sat_3.0, fvp_unsat_1.0, fvp_unsat_2.0, fvp_unsat_3.0.

Table 2. Performance comparison of all strategies over data sets, two metrics (M) are presented: number of solved instances (S) and total runtime (T, in seconds). Average solver behaviour (a randomly picked solver) is presented as *Rand.*. Runtime cutoff is 5400 second. Unsolved instances are considered to contribute 5400 seconds to the total runtime.

Strategy	*M*	*bmc*		*velev*		*crypto*		*random*		*Total*
		SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	
Luby-32	S	192	220	53	64	67	300	427	561	1884
	T	143.26	98.68	37.86	72.30	138.02	16.65	90.34	96.60	693.74
Luby-512	S	196	**226**	59	64	71	300	431	564	1911
	T	132.90	**93.48**	31.72	71.48	141.19	14.30	74.89	86.44	646.38
Fix-512	S	142	187	58	62	50	300	414	558	1771
	T	185.73	134.17	32.66	73.29	164.78	15.94	105.10	103.56	815.24
Fix-4096	S	199	213	59	64	73	300	431	563	1902
	T	138.67	110.52	33.02	69.80	136.46	13.57	72.63	83.30	657.97
Fix-16384	S	191	212	**61**	65	69	300	432	569	1899
	T	157.68	119.88	**24.77**	**68.16**	147.86	13.10	65.37	72.34	669.17
Geom-1.1	S	189	213	59	64	62	300	432	571	1890
	T	127.93	104.83	29.94	69.02	149.69	13.96	68.18	79.44	642.99
Geom-1.5	S	128	174	54	62	84	300	**439**	573	1814
	T	213.27	155.04	37.81	72.62	130.86	12.93	**54.00**	**70.87**	747.39
Nest-1.1	S	192	218	**61**	65	76	300	423	566	1901
	T	137.64	107.69	32.49	69.76	134.94	15.65	92.27	94.87	685.31
Nest-1.5	S	179	215	58	63	**88**	300	434	**574**	1911
	T	152.67	100.12	33.82	70.29	**122.52**	**11.61**	69.60	75.93	636.56
Rand.	S	178.67	208.67	58	63.66	62.77	300	429.22	566.56	1862.04
	T	154.42	113.82	32.68	70.74	140.71	14.19	76.93	84.82	666.98
LMPick	S	**203**	221	**61**	**66**	72	300	435	571	**1929**
	T	**124.57**	97.53	28.09	68.98	138.17	12.02	68.00	80.86	**618.23**

In terms of problem solved, *LMPick* performs better than any given restart strategy in two of the cases, and performs on par with the optimal strategy in two other cases. The "Total" column shows that for these data sets, *LMPick* would solve more instances in less time than any single restart strategy.

For the *bmc* data set, *LMPick* performs significantly better than any other restart scheme for satisfiable instances. It solves 4 instances more than the best strategy (*Static-4096*), and the total runtime shows 19.33% improvement over the average *Random-pick* strategy. For unsatisfiable instances, it performs worse. Although it is second amongst the strategies, it is clearly worse than *Luby-512* that solves 6 more instances. Comparing the standard deviation of these two data sets in Table 1 does not explain this descripency in performance. Both show large, almost similar, coefficients (sat's standard deviation is 154.42, and its variation coefficient is 0.171 while unsat's standard deviation is 113.82 and its variation coefficient is 0.1633), this indicates a high potential of improvement for both data sets. We conjecture that the reason for the poorer performance lies

Table 3. Accuracy results for the satisfiability classifier. Figures represent the percent of correctly classified instances.

Class	*bmc*	*velev*	*crypto*	*rand*
SAT	100.00%	97.22%	99.28%	82.49%
UNSAT	83.54%	82.85%	100.00%	94.51%
ALL	91.72%	88.70%	99.77%	89.31%

Table 4. Performance of *LMPick*'s chosen restart strategy (s_b) in comparison with other strategies. The figures represent the percent of strategies that outperform s_b for a given data set.

Class	*bmc*	*velev*	*crypto*	*rand*
SAT	30.18%	35.58%	32.24%	39.29%
UNSAT	32.01%	37.05%	14.77%	39.61%

in the fact that both the classifier, and the runtime models are more accurate for *bmc-sat* than for *bmc-unsat*.

Table 3 presents the performance of the satisfiability classifier. The figures in the table represent the percent of instances that were predicted correctly. It is important to note that since we did not apply a "winner takes all" approach, in some of the cases where the classification was correct but not with 100% certainty, the wrong model was also considered. We can see that for *bmc*, satisfiable instances are classifed correctly every time, while unsatisfiable ones are classified correctly only 83.54% of the times. In order to check the influence of these classifaction errors on the performance of *LMPick* for this dataset, we ran another experiment, where we used a classification oracle. *LMPick*'s performance is improved, and it solved 224 instances, which are 3 more than the original results. This shows the effect of a good classification technique over the overall performance.

In order for *LMPick* to enhance the solver's performance, runtime estimation models need to perform well. Nevertheless, it is not crucial that each model's prediction is accurate. The important factor is the relative order of these predictions. We would prefer the chosen strategy (s_b) to be amongst the best strategies for that instance. Table 4 demonstrates the quality of the chosen strategy s_b. The table presents the percentage of strategies that outperform s_b. This table shows that in all cases s_b is better than picking a random strategy, and that for the *crypto-unsat* instances it performs very well.

The difference in the performance on the *crypto-sat* data set is not easily explained by the data presented so far. While both the classifier and runtime prediction models perform better than on the *velev-sat*, the overall performance is worse. We conjecture that the cause of this difference is the differing likelihood of an instance being solved by multiple strategies.

Figure 4 compares the 3 non-random satisfiable data sets. In Fig. 4(a) bars represent the percent of instances in the data set that were solved within the cutoff time by each number of strategies. There is a clear difference between

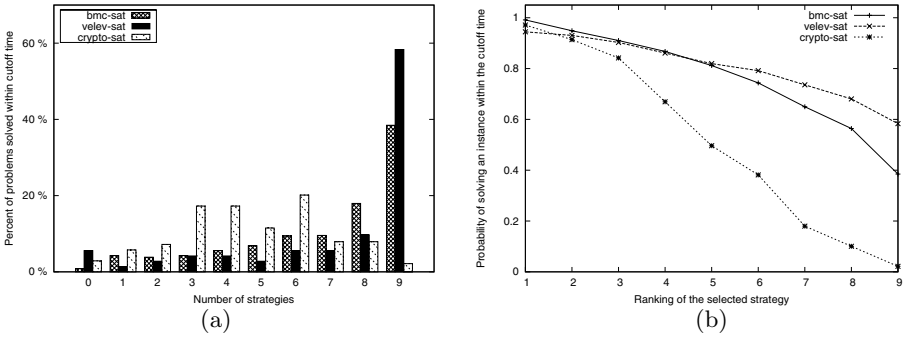


Fig. 4. A comparison of strategies' performance on three satisfiable data sets. On the left, bars represent the percent of instances that are solved (within cutoff time) by each number of strategies, when 0 means all instances that cannot be solved by any of the strategies and 9 means all instances that are solved by all strategies. On the right the plot shows the probability of solving a randomly picked instance (within cutoff time) as a function of the quality of the selection decision, when 1 means the strategy selected was the best one and 9 means it was the worst.

crypto-sat and the other 2 data sets. While for both industrial verification based data sets, most instances were solved by the majority of strategies, for *crypto-sat* many of the instances are solved by a small set of strategies. The effect of this difference is demonstrated in Fig. 4(b). This plot presents the probability of a randomly picked instance being solved within the cutoff time given the quality of selected strategy. From left to right, the picked strategy shift from best to worse. If *LMPick* picks one of the the two best strategies, the probability of all three data sets is quite similar, but when the chosen strategy is 3rd or 4th, the probability of solving a *crypto-sat* instance drops significantly compared to the other two. Many instances in the *crypto-sat* data set are only solved within the cutoff by a small subset of strategies, making this data set harder as a sub-optimal selection is likely to lead to a timeout.

5 Conclusions and Future Work

Restart strategies have an important role in the success of DPLL style SAT solvers. The performance of different strategies varies over different benchmark families. We harness machine learning to enhance the performance of SAT solvers. We have presented *LMPick*, a technique that uses both satisfiability classification and solver runtime estimation to pick promising restart strategies for instances. We have demonstrated the effectiveness of *LMPick* and compared its results with the most commonly used restart strategies. We have established that in many cases *LMPick* outperforms any single restart strategy and that it is never worse than a randomly picked strategy. We have also discussed the influence of different components of *LMPick* on its performance.

While universal restart strategies are more commonly used than dynamic ones, dynamic strategies are getting more attention lately. An interesting continuation to this work would be to use machine learning to develop a fully dynamic restart strategy. Such a strategy could use unsupervised machine learning algorithm to develop a dynamic restart policy for a benchmark family of problems. Restart strategies are not the only aspect of SAT solving influencing performance on different benchmark families. Machine learning can be also used to tune other parameters that govern the behavior of modern DPLL based solvers, namely, parameters that are commonly set manually as a result of a trial and error process, such as decision heuristic parameters, clause deletion policy, etc.

Acknowledgements

NICTA is funded by the Department of Broadband, Communications and the Digial Economy, and the ARC through Backing Australia's Ability and the ICT Centre of Excellence program.

References

1. Biere, A.: Adaptive Restart Strategies for Conflict Driven SAT Solvers. In: Proc. of the 11th Int. Conf. on Theory and Applications of Satisfiability Testing (2008)
2. Biere, A.: PicoSAT Essentials. Journal on Satisfiability, Boolean Modeling and Computation 4, 75–97 (2008)
3. Bregman, D., Mitchell, D.: The SAT solver MXC (version 0.75). Solver Description for the SAT Race 2008 solver competition (2008)
4. Devlin, D., O'Sullivan, B.: Satisfiability as a Classification Problem. In: Proc. of the 19th Irish Conf. on Artificial Intelligence and Cognitive Science (2008)
5. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. of the 6th Int. Conf. on Theory and Applications of Satisfiability Testing (2003)
6. Eibach, T., Pilz, E., Völkel, G.: Attacking Bivium Using Using SAT Solvers. In: Proc. of the 11th Int. Conf. on Theory and Applications of Satisfiability Testing (2008)
7. Frost, D., Rish, I.: Summarizing CSP hardness with continuous probability distributions. In: Proc. of the 14th National Conf. on Artificial Intelligence (1997)
8. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. In: Proc. of Design Automation and Test in Europe (2002)
9. Gomes, C.P., Selman, B., Kautz, H.: Boosting Combinatorial Search through Randomization. In: Proc. of the 15th National Conf. on Artificial Intelligence (1998)
10. Haim, S., Walsh, T.: Online Estimation of SAT Solving Runtime. In: Proc. of the 11th Int. Conf. on Theory and Applications of Satisfiability Testing (2008)
11. Huang, J.: The effect of restarts on the efficiency of clause learning. In: Proc. of the 20th Int. Joint Conf. on Artificial Intelligence (2007)
12. Huang, J.: A Case for Simple SAT Solvers. In: Proc. of the 13th Int. Conf. on Principles and Practice of Constraint Programming (2007)
13. Hutter, F., Hamadi, Y., Hoos, H., Leyton-Brown, K.: Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms. In: Proc. of the 12th Int. Conf. on Principles and Practice of Constraint Programming (2006)

14. Kautz, H., Horvitz, E., Ruan, Y., Gomes, C., Selman, B.: Dynamic Restart Policies. In: Proc. of the 18th National Conf. on Artificial Intelligence (2002)
15. Krishnapuram, B., Figueiredo, M., Carin, L., Hartemink, A.: Sparse Multinomial Logistic Regression: Fast Algorithms and Generalization Bounds. IEEE Transactions on Pattern Analysis and Machine Intelligence 27, 957–968 (2005)
16. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. In: Proc. of the 2nd Israel Symp. on the Theory and Computing Systems (1993)
17. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proc. of the 38th Design Automation Conference (2001)
18. Nudelman, E., Leyton-Brown, K., Hoos, H.H., Devkar, A., Shoham, Y.: Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In: Wallace, M. (ed.) CP 2004, vol. 3258, pp. 438–452. Springer, Heidelberg (2004)
19. Ruan, Y., Horvitz, E., Kautz, H.: Restart Policies with Dependence among Runs: A Dynamic Programming Approach. In: Van Hentenryck, P. (ed.) CP 2002, vol. 2470, p. 573. Springer, Heidelberg (2002)
20. Ruan, Y., Horvitz, E., Kautz, H.: Hardness-aware restart policies. In: The 18th Int. Joint Conference on Artificial Intelligence: Workshop on Stochastic Search (2003)
21. Ryan, L.: Efficient algorithms for clause learning SAT solvers. Master thesis, Simon Fraser University, School of Computing Science (2004)
22. Ryvchin, V., Strichman, O.: Local Restarts. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 271–276. Springer, Heidelberg (2008)
23. Walsh, T.: Search in a Small World. In: Proc. of the 12th Int. Joint Conference on Artificial Intelligence (1999)
24. Wu, H., van Beek, P.: On Universal Restart Strategies for Backtracking Search. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 681–695. Springer, Heidelberg (2007)
25. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based Algorithm Selection for SAT. Journal of Artificial Intelligence Research 32, 565–606 (2008)
26. Xu, L., Hoos, H., Leyton-Brown, K.: Hierarchical Hardness Models for SAT. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 696–711. Springer, Heidelberg (2007)

Instance-Based Selection of Policies for SAT Solvers*

Mladen Nikolić, Filip Marić, and Predrag Janičić

Faculty of Mathematics, University of Belgrade,
Belgrade, Studentski Trg 16, Serbia
{nikolic,filip,janicic}@matf.bg.ac.rs

Abstract. Execution of most of the modern DPLL-based SAT solvers is guided by a number of heuristics. Decisions made during the search process are usually driven by some fixed heuristic policies. Despite the outstanding progress in SAT solving in recent years, there is still an appealing lack of techniques for selecting policies appropriate for solving specific input formulae. In this paper we present a methodology for instance-based selection of solver's policies that uses a data-mining classification technique. The methodology also relies on analysis of relationships between formulae, their families, and their suitable solving strategies. The evaluation results are very good, demonstrate practical usability of the methodology, and encourage further efforts in this direction.

1 Introduction

The propositional satisfiability problem (SAT) is one of the fundamental problems in computer science. It is the problem of deciding if there is a truth assignment under which a given propositional formula (in conjunctive normal form) evaluates to true. SAT was the first problem proved to be NP-complete [Coo71] and it still has a central position in the field of computational complexity. SAT problem is also very important in many practical domains such as electronic design automation, software and hardware verification, artificial intelligence, and operations research. Thanks to recent advances in propositional solving technology, *SAT solvers* (procedures that solve the SAT problem) are becoming a tool for attacking more and more practical problems.

A number of SAT solvers have been developed. The majority of state-of-the-art complete SAT solvers are based on the branch and backtrack algorithm called Davis-Putnam-Logemann-Loveland or the DPLL algorithm [DP60, DLL62]. Spectacular improvements in the performance of DPLL-based SAT solvers achieved in the last few years are due to (i) several conceptual enhancements on the original DPLL procedure, aimed at reducing the amount of explored search space (e.g., *backjumping*, *conflict-driven lemma learning*, *restarts*), (ii) better implementation techniques (e.g., *two-watched literals* scheme for unit propagation), and (iii) smart heuristic components (which we focus on in this

* This work was partially supported by Serbian Ministry of Science grant 144030.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 326–340, 2009.

work). These advances make possible to decide the satisfiability of industrial SAT problems with tens of thousands of variables and millions of clauses.

Complex *policies*, heuristics that guide the search process, represent important parts of modern SAT solvers and are crucial for solver's efficiency. These include policies for literal selection, for determining the clause database size, for choosing restart points, etc. Specific policies are usually *parameterized* by a number of numerical and categorial parameters. Single policy with different parameter values can be treated as different policies. SAT solving process is completely determined (up to randomized choices) only when all its heuristic policies are set. Selected combinations of policies specify the *solving strategy* (or simply *strategy)*.

Typically, every SAT solver uses a predetermined, hard-coded strategy and applies it on all its input formulae. However, in recent times, SAT solvers tend to implement multiple policies and the question arises as to how to choose a strategy that would give good performance for a specific SAT instance. Addressing this question is of crucial importance because the solving time for the same input formula can vary for several orders of magnitude depending on the solving strategy used. The problem of adapting a SAT solver to the input formula has been addressed for the first time only recently. Our approach significantly differs from the only existing related approach we are aware of (as discussed in Sect. 5).

Propositional formulae can be clustered in families of formulae by their origin — industrial problems (e.g., FPGA routing), manually crafted problems (e.g., graph coloring, Hanoi towers), or random generated problems (e.g., k-SAT). It is interesting to explore the behaviour of different policies and solving strategies on families of formulae. The important question is whether one strategy shows the same or similar behaviour on similar formulae. If this is the case, and if one can automatically guess a family to which a given formula belongs, then this could be used for selecting an appropriate strategy for this particular formula. To implement this approach, one needs (i) a technique for classifying formulae based only on their syntax; (ii) information about behaviour of different policies on various families of formulae.

The main message of this work is that intelligent selecting of solving policies, based on the syntax of the input formula, can significantly improve efficiency of a SAT solver. The proposed methodology will not lead to optimal performance on each input formula, but the solving performance will be significantly improved in average on multiple input formulae. Here, by improving efficiency of a SAT solver we mean increasing the number of formulae solvable within some time limit and decreasing the solving time.

The proposed methodology relies on several hypotheses that will be investigated in the rest of the paper:

(H1) Formulae of the same family (i.e., of similar origin) share some syntactical properties that can be used for automated formula classification;

(H2) For each family of formulae there is only a small number of solving strategies that are appropriate — that show better performance on formulae belonging to that family then all other available strategies.

(H3) For formulae that are syntactically similar, the best strategies are also (in some sense) similar.

If the above hypotheses hold, then our methodology will be practically applicable. Namely, if the formula is correctly classified then it has a good chance to be solved by a solving strategy suitable for a family that the formula belongs to. However, even if the formula is misclassified, it will be solved using a strategy similar to the optimal one.

The rest of the paper is organized as follows: in Sect. 2, a brief background information on SAT problem, SAT solvers, and their heuristic components is given. In Sect. 3, the proposed methodology is described. The experimental results are given in Sect. 4. In Sect. 5 related work is discussed. In Sect. 6 final conclusions are drawn and some directions of possible further work are discussed.

2 Background

Most of today's state-of-the-art solvers are complex variations of the DPLL procedure. In the rest of the paper, we shall assume that the reader is familiar with the modern SAT solving techniques. More on these topics can be found, for example, in [NOT06, KG07, Mar08, GKSS07]. Although modern SAT solvers share common underlying algorithms and implementation techniques, their operation is guided by a number of heuristic policies that have to be selected in order to define solving strategies. The most important heuristic policies determine: (i) which literals to choose for branching, (ii) when to apply restarting, and (iii) when to forget some clauses that are learnt during the solving process. In the rest of this section, policies that were varied in our experiments will be described.

Literal selection policies. During the DPLL backtrack-search, literals used for branching should be somehow selected. This is the role of *literal selection policies*. Most literal selection policies separately select a variable v (by using a *variable selection policy*) and only then choose its polarity, i.e., choose if it should be negated or not (by using a *polarity selection policy*).

Some variable selection policies are the following[1]:

VS_{random} — This policy randomly chooses a variable among all variables of the initial formula that are not defined in the current valuation, i.e., assertion trail.

$\mathsf{VS}^{b,d,init}_{VSIDS}$ — The goal of this policy (introduced in the solver CHAFF [MMZ+01]) is to select a variable that was active in recent conflicts. In order to implement this, an activity score is assigned to each variable. On every conflict, the scores of the variables that occur in the conflict clause are bumped, i.e., increased by a *bump factor* given by the parameter b. Also, during the conflict analysis process, on each resolution step all variables that occur in the

[1] The policy names will be printed in subscripts and their parameters in superscripts.

explanation clause are bumped. To stimulate recent conflicts, on each conflict all the scores are decayed, i.e., decreased by a *decay factor* given by the parameter d.

An important aspect of the VSIDS variable selection policy is how to assign initial scores to variables. If the parameter *init* has the value ZERO, scores of all variables are set to zero, hence all variables have the same chance to be selected. If the parameter *init* has the value FREQ, the initial score of each variable is set to its number of occurrences in the initial formula F_0.

$\mathsf{VS}^{p}_{random} \circ \mathsf{VS}_{x}$ — This compound policy chooses a random variable with probability p and otherwise uses a given policy here denoted by VS_x.

Some polarity selection policies are the following:

$\mathsf{PS}_{positive}$ — Always selects a non-negated literal.

$\mathsf{PS}_{negative}$ — Always selects a negated literal.

PS^{p}_{random} — A random selection which chooses a non-negated literal with probability p.

$\mathsf{PS}^{init}_{polarity\_caching}$ — When using this policy (introduced in the solver RSAT [PD07] as *phase caching*), a preferred polarity is assigned to each variable and it is used for polarity selection. Whenever a literal is asserted to the current assertion trail (either as a decision or as a propagated literal), its polarity defines the future preferred polarity of its variable. When a literal is removed from the trail (during backjumping or restarting) its preferred polarity is not changed. If the parameter *init* has the value POS, then initial polarities of all variables are positive, it has the value NEG then they are set to negative, and if it has the value FREQ, then preferred polarity of each variable is set to the polarity which is more frequent of the two in the initial formula.

Restart policies. Restart policies determine when to apply restarting. Most restart policies are based on conflict counting. On each conflict, the counter is increased. Restarting is applied whenever the counter reaches a certain threshold value. When this happens, the counter is reset and a new threshold is selected according to some specific policy. Some possible restart policies are the following:

$\mathsf{R}_{no\_restart}$ — Restarting is not applied.

$\mathsf{R}^{c_0,q}_{minisat}$ — The initial threshold value is set to c_0 and the threshold values form a geometric sequence with a quotient q [ES04].

R^{m}_{luby} — The threshold values are elements of the Luby series [LSZ93] multiplied by a positive integer m, while the Luby series is:

$$t_i = \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1} & \text{if } 2^{k-1} \leq i \leq 2^k - 1 \end{cases}$$

Its first few elements for $m = 1$ are $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, ...$

$\mathsf{R}^{c_0,q}_{picosat}$ — In this policy (introduced by the solver PICOSAT [Bie08]), restarts are controlled by two geometric sequences of threshold values — inner and

outer, both with an initial member c_0 and a quotient q. Restarting is applied when the number of the conflicts reaches the current inner threshold value, and then the inner threshold value advances to the next element of the sequence. When the inner threshold exceeds the current outer threshold value, it is reset to the initial value c_0, and the outer threshold advances.

3 Methodology

Our methodology, when applied to a given SAT solver, selects an appropriate solving strategy (i.e., combination of policies) for each input formula. The proposed methodology can be applied to any DPLL-based SAT solver, provided it supports multiple policies. In our experiments, the ArgoSAT solver[2] was used since it implements a large number of policies and since its modular architecture allows easy modification of existing and implementation of new policies [Mar09].

The overall methodology consists of two phases.

Training phase. This phase consists of systematic solving of all formulae from a representative corpus, by using all candidate solving strategies. This allows selecting the best solving strategy (the one that solves the most formulae) for each family of formulae from the corpus. *Profiles* of all formulae from the corpus (i.e., their representation suitable for classification) are also computed in this phase.

Exploitation phase. In this phase, the family of a given formula is guessed and the strategy that showed the best results on that family during the training phase is used for its solving.[3] After the training, the system implementing the methodology can be applied both to the formulae from the training corpus and to some other formulae.

Several issues still need to be addressed, as discussed below.

The choice of candidate solving strategies. In our case candidate strategies are defined as all $(60 = 3 \cdot 5 \cdot 4)$ possible combinations of the given policies. They are listed in Table 1. Apart from the policies that are subject to automatic selection, some important policies (e.g., forget and conflict analysis) are fixed.[4] We do not claim that some other policies could not give better performance. Some of the considered policies are even expected to be inferior. However, we are proposing a general methodology that can be used with any input set of policies.

[2] http://argo.matf.bg.ac.rs/software/

[3] We also tested an alternative approach that does not use information on families. In that approach, k ($k \geq 1$) formulae that are most similar to the input formula are detected. Then, a solving strategy that occurs most frequently among l ($l \geq 1$) best strategies for each detected formula is chosen. This alternative approach will not be discussed in more details because it gave inferior results.

[4] A MiniSAT-style forget policy [ES04] and 1UIP technique [ZMMM01] for conflict analysis are used.

Table 1. Overview of policies used in our experiments

Variable selection	VS_{random}, $\mathsf{VS}_{VSIDS}^{1.0,\,1.0/0.95,\,\text{FREQ}}$, $\mathsf{VS}_{random}^{0.05} \circ \mathsf{VS}_{VSIDS}^{1.0,\,1.0/0.95,\,\text{FREQ}}$
Polarity selection	PS_{POS}, PS_{NEG}, $\mathsf{PS}_{random}^{0.5}$, $\mathsf{PS}_{polarity\_caching}^{\text{NEG}}$, $\mathsf{PS}_{polarity\_caching}^{\text{FREQ}}$
Restart policies	$\mathsf{R}_{no\_restart}$, $\mathsf{R}_{minisat}^{100,1.5}$, R_{luby}^{512}, $\mathsf{R}_{picosat}^{100,1.5}$

The choice of corpus of formulae for training and evaluation. In our experiments (both during the training and for testing), the corpus from the *SAT competition* in 2002 was used. It consists of a large number of families of formulae, with many families containing formulae of various difficulty. The total number of formulae in this corpus is 1964, and we clustered them into 39 families, mostly just by following the directory structure. Since this corpus was systematically solved in the training phase, it can be used both for testing of hypotheses and for thorough analysis of the proposed methodology. For testing of a generalization power of our methodology on a different corpus, the *SAT competition* corpus from 2007 was used. Namely, out of 906 formulae in this corpus, only 12 of them also belong to the corpus from 2002. In addition, the two corpora include significantly different families (although overlapping exists). Since the SAT2007 corpus was used only for evaluation of our resulting solving system, we do not consider its partitioning into families.

The choice of relevant features of propositional formulae. In order to measure the syntactic similarity of propositional formulae (which is necessary for classification), the formulae were represented by using the first 33 features used in [XHHLB08]. These are features that can be calculated in short time. They include the number of clauses c and variables v in the input formula, their ratio $\frac{c}{v}$, fraction of binary, ternary, and Horn clauses, node degree statistics for variable nodes in variable-clause graph like mean, variation coefficient, minimum, maximum, and entropy, etc. The vectors of these features are called the *formula profiles* or simply *profiles*.

The choice of methods for classification of propositional formulae. For classification, the *k-nearest neighbour* algorithm was used. When given an unknown instance, this algorithm selects the class that contains the most of the k instances from the training corpus that are closest to the given one. A number of distance functions between the profiles were used [TJK06]. For instance:

$$d_1(\mathcal{P}', \mathcal{P}'') = \sqrt{\sum_i (\mathcal{P}'_i - \mathcal{P}''_i)^2} \quad d_2(\mathcal{P}', \mathcal{P}'') = \sum_i \left(\frac{\mathcal{P}'_i - \mathcal{P}''_i}{\sqrt{|\mathcal{P}'_i \mathcal{P}''_i)| + 1}} \right)^2$$

$$d_3(\mathcal{P}', \mathcal{P}'') = \sum_i \frac{|\mathcal{P}'_i - \mathcal{P}''_i|}{\sqrt{|\mathcal{P}'_i \mathcal{P}''_i| + 1}} \quad d_4(\mathcal{P}', \mathcal{P}'') = \sum_i \left(\frac{\mathcal{P}'_i - \mathcal{P}''_i}{\sqrt{|\mathcal{P}'_i \mathcal{P}''_i| + 10}} \right)^2$$

where $\mathcal{P}'$ and $\mathcal{P}''$ are instance profiles.

4 Experiments and Evaluation

Experiments described in this section test the hypotheses that our approach relies on (given in Sect. 1), and also demonstrate a good overall quality of our methodology.

Training Phase. During the training phase, the cutoff time for solving one formula by one strategy was set to 600s. It would be interesting to consider higher cutoff times as well, but this choice was made with regard to available computational resources. Since shuffling of clauses and variables of a formula can lead to big differences in its solving time (up to an order of magnitude), the solving times associated with the formulae were calculated in the following way. For each formula, the original and one shuffled variant were solved. If both variants of the formula were solved within the time limit, the arithmetic mean of their solving times was associated to the formula. If either variant was not solved within the cutoff time limit, the formula was considered to be unsolved. The SAT solver was used on all the formulae from the extended corpus, for all 60 strategies. The total number of calls to the SAT solver was 235680 ($= 1964 \cdot 60 \cdot 2$). The experiments were conducted on an IBM Cluster 1350 cluster computer with 32 processors. The total processor time used was around 1010 days.

Along with solving the formulae, their profiles were computed. The average profile computation time was 0.39s per formula.

4.1 Testing Hypotheses

Hypothesis (H1). The first hypothesis is that formulae from the same family share syntactical properties that can be used for automated formula classification. In the k-nearest neighbours method, values $1, 3, 5, 7$ were used for k. The best results were obtained for $k = 1$ with the distance function d_3[5]. The precision, a ratio between the number of correctly classified formulae and the total number of formulae classified, was 98.5%. The arithmetic mean of precisions for individual families[6], was 89.4%. To avoid evaluating on the same data that was used for training, both statistics were estimated using the *leave-one-out procedure*. This procedure consists of removing formula from the corpus, computing the relevant statistic on the rest of the corpus, and returning the formula into the corpus. This is done for all formulae. The obtained values of the statistic were averaged at the end to give the final estimate of the statistic on given data.

The results of the classification are outstanding (especially keeping in mind a rather large number of classes — 39) and show that the first hypothesis of the methodology is sound. Since the average profile computation time for a formula is 0.39s and the classification time of a known profile is less than 0.01s, this

[5] Therefore, in all experimental results in the rest of the paper, this distance function will be assumed.

[6] Precision alone is not reliable in cases when some families are much larger than the others (which is the case with the SAT2002 corpus), so a high precision on large classes can hide a low precision on small classes.

approach to classification is practically usable for our purposes and may have applications in other domains too.

Hypothesis (H2). The second hypothesis of our methodology is that there is a small number of strategies for each family of formulae that show better performance on formulae belonging to that family then all other available strategies. To check this hypothesis, for each strategy and for each family of formulae a percentage of formulae for which that strategy was better then any other strategy was calculated. The results are shown in a graphical form in Fig. 1. In the left part of the figure, darker shades correspond to higher values, and lighter to lower values. The highest value (i.e., percentage) in the table is 30, and the lowest value is 0. In the right part of the figure, normalized entropies for families are shown. For simplicity, the results are shown only for families with at least 10 formulae that were solved by at least one strategy. The figure shows that there is no family with a dominantly best strategy. However, the presented matrix is sparse and the average normalized entropy for all families is 0.39 — hence, for each family there is a rather small set of good strategies and therefore, the second hypothesis can be considered to be justified.

These results also reveal the quality of some strategies. For instance, 15 empty columns correspond to strategies with VS_{random} variable selection policy, which suggests a poor performance of this policy.

Hypothesis (H3). The third hypothesis of our methodology is that the best solving strategies for syntactically similar formulae are also similar. Syntactical similarity between formulae is already defined by the choice of profiles and the distance function (d_3) from Sect. 3. On the other hand, similarity between strategies can be defined using the *edit distance* over strategies:

$$d_c(s_1 s_2 s_3, t_1 t_2 t_3) = \sum_{i=1}^{3} c(s_i, t_i)$$

where $s_1 s_2 s_3$ and $t_1 t_2 t_3$ are triples of policies that determine strategies and $c(s_i, t_i)$ are non-negative numerical costs of switching from policy s_i to policy t_i.

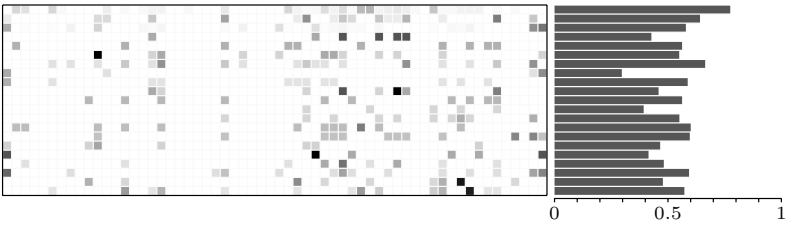


Fig. 1. The left part shows percentages of formulae from a family for which a strategy is better than any other (columns correspond to strategies and rows correspond to families). The right part shows normalized entropy values for families.

To analyze the correlation between similarity of formulae and similarity of their corresponding best strategies, for each two formulae f_1 and f_2 from the corpus, with best strategies c_1 and c_2 respectively, values $\log d_3(f_1, f_2)$ and $d_c(c_1, c_2)$ were calculated. Then, the Pearson correlation coefficient between these sets of values was calculated. The costs in the function d_c were manually tuned to achieve a maximal correlation coefficient. This procedure was legitimate and it is closely related to the following question: for which group of policies (restart, variable selection, literal selection) optimal choices differ the least for syntactically similar formulae? Only formulae solved in more than 2s were included in the calculation.[7]

The calculated correlation coefficient was 0.51 with the p-value less then 0.001. This can be considered a moderate, but important correlation, keeping in mind the small number of policies that were used in the experiments and the inherent instability of the SAT solving process. Hence, we can consider the third hypothesis to be justified.

Magnitudes of costs in the function d_c, suggest that for syntactically similar formulae, the optimal restarting policy varies the least (thus being in some sense the strongest common characteristic of syntactically similar formulae) and the polarity selection policy varies the most.

4.2 Exploitation Phase

Evaluation of Strategy Selection Method on the SAT2002 Corpus. For evaluation purposes the proposed strategy selection method was compared to (i) the *"best-fixed"* strategy selection method which always uses the best fixed candidate strategy[8], and to (ii) the *"oracle"* (idealized) strategy selection method which uses the best candidate strategy for each specific formula. The first one is a fair choice for the lower bound of required performance and the second one represents the upper bound for performance because it gives the best possible performance over the set of candidate strategies on the SAT2002 corpus.

As the main measures of the overall quality of a strategy selection method the total number of formulae that it solved and its median solving time were used. There are strong reasons to base the evaluation on median instead on mean and total solving time. First, one cannot account for the censored data — solving times over the cutoff limit. If one chooses not to include these data in the calculation then the mean and total solving time show preference for solvers that solve less formulae because even if a hard formula is solved its solving time is typically near the cutoff limit, and thus raises the mean and the total solving time. On the other hand, if at least half of the formulae were solved, the median time can be calculated. Also, median time is known to be less sensitive to outliers.

To estimate the performance on the SAT2002 corpus, similarly as in the leave-one-out procedure, the solving time of each formula was computed for the

[7] Trivial formulae cannot discriminate between good and bad strategies. Also, variation of their solving time due to processor context switching is larger relative to their solving time.

[8] In our case the best strategy was $(\mathsf{VS}^{1.0,\,1.0/0.95,\,\mathrm{FREQ}}_{VSIDS}, \mathsf{PS}^{\mathrm{NEG}}_{polarity\_caching}, \mathsf{R}^{100,1.5}_{minisat})$.

Table 2. Comparison of strategy selection methods on SAT2002 corpus

Method	Number of solved formulae	Median solving time
"Best fixed"	1073	207.4s
Proposed	1135	92.6s
"Oracle"	1187	45.8s

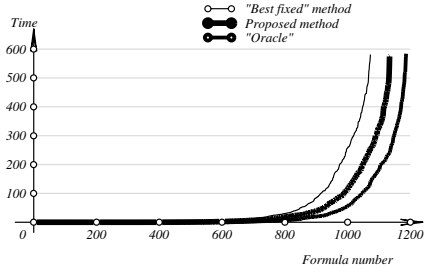


Fig. 2. Sorted solving times for different ways of choosing the strategy. Ordinal numbers in the sequence of the sorted solving times of the formulae are shown on the X axis. Solving times are shown on the Y axis.

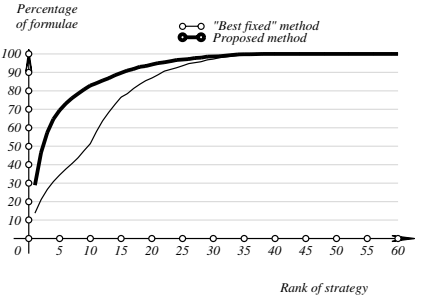


Fig. 3. Cumulative distribution function of ranks of chosen strategies for the "best fixed" method and the proposed method. Ranks of strategies are shown on the X axis, while percentage of formulae for which strategies of that or smaller rank are chosen is shown on the Y axis.

strategy selected based on the results of the training phase, but with that formula excluded from the corpus. Table 2 shows the results for the two referent methods and for the one proposed. Important results were achieved, especially keeping in mind a rapid growth of the sorted solving times of formulae for all three methods, as shown in Fig. 2. The differences in the numbers of the solved formulae for all three methods seem small, but would be much higher if numerous easy formulae were not taken into account.

Given a strategy selection method, it is important to consider how often it chooses best, good or bad strategies (especially when compared to some other selection methods). For each formula, candidate strategies can be sorted in ascending order according to their corresponding solving times. Each strategy can be given a rank according to its position in such sequence. A strategy with the rank 1 is the most desirable for that specific formula. In Fig. 3 we present the cumulative distribution function of the ranks of chosen strategies for the proposed and for the "best fixed" method. The figure shows that the proposed method chooses good strategies much more often than the "best fixed" method.

The histogram in Fig. 4 shows the number of formulae solved by the proposed method in some percentage of a referent time, up to 200%. As a referent time we use the solving time obtained by the "best fixed" method. Only formulae

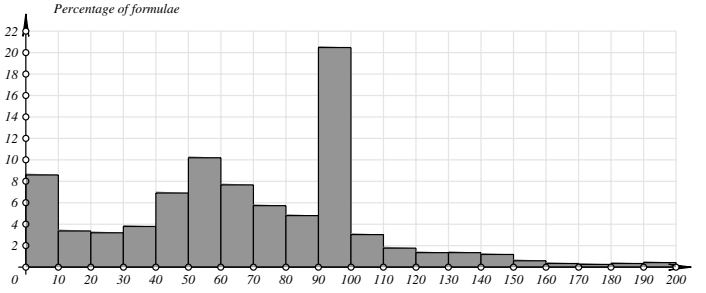


Fig. 4. Histogram of the number of formulae solved by the proposed method in some percentage of the referent time. Percentage of the referent time is shown on the X axes, while the number of formulae solved in that percentage of the referent time is shown on the Y axis.

that were solved either way were considered. 48 formulae are solved in more than 200% of the referent time. There are 74 formulae that were solved by the proposed method but not by the "best fixed" method, but only 12 that were solved by the "best fixed" method and not by the proposed method. It can be observed that much more formulae were solved faster than slower, compared to the "best fixed" method.

These results show that the main thesis of this work — that intelligent choosing of the solver's strategy based on the syntax of the input formula, can significantly improve efficiency of a SAT solver — is true.

Evaluation of ArgoSmArT *system on SAT2007 corpus.* Based on the methodology described above, we implemented a SAT solving system ArgoSmArT (on top of the ArgoSAT solver).

For an additional evaluation of the proposed methodology, we used the SAT2007 corpus and showed that performance improvement achieved on one corpus is present on a different corpus too.

The formulae from the SAT2007 corpus are much harder then the ones from the SAT2002 corpus and the median time cannot be calculated, since in cutoff time of 600s less than a half of the formulae can be solved. Thus, we present 20-th percentile of the solving times[9]. The results of the comparison of ArgoSmArT to its base solver ArgoSAT are shown in Table 3 (ArgoSAT used the best fixed strategy detected in the experiments on the SAT2002 corpus).

Notice that the performance improvement achieved on the SAT2002 corpus in the number of solved formulae is also present on the SAT2007 corpus. The improvement in the solving time is also significant. As said above, these two corpora share only 12 formulae, and only one of them was solved by the ArgoSmArT

[9] 20-th percentile of the solving times is the value that splits the sorted solving times in two parts, the lower one having 20% of the total number of values.

Table 3. Results of comparison between ArgoSAT and ArgoSmArT on the SAT2007 corpus

System	No. of solved formulae	20-th percentile of solving time
ArgoSAT	219	311.6s
ArgoSmArT	239	249.5s

within 600s, so the improvement cannot be attributed to overlapping of the training and the test set.

5 Related Work

Hypotheses like H2 and H3 are already discussed as a basis for instance-based solving of the algorithm selection problem [SM08]. Algorithm selection for constraint satisfaction problems (CSP) based on performance prediction is described in [LL98]. A reinforcement learning based approach to choose variable selection policy for CSP, with only preliminary results is described in [XSS09]. In quantified boolean formulae (QBF) solving, multinomial logistic regression was successfully used for dynamic, online selection of variable selection policies [SM07]. Strategy selection for MiniSAT based on neural networks that gave limited results is described in [Kib07].

Features used for classification in this paper are first described in [NBH+04] for prediction of a solver's running time and were later used in SATzilla system [XHHLB08]. SATzilla is the system that uses linear regression predictions of solver running times to select one of its component solvers for solving an input formula. As reported in [XHHLB08] for the corpus SAT2007, evaluation on the random category of SAT instances, demonstrated a significant improvement in running time. Average running time for SATzilla system was around 90s, while its best component solver average was 290s. On the crafted category, SATzilla average was around 150s, compared to its best component solver average of 280s. For the industrial category, this improvement was smaller, but still significant — 260s compared to 330s. In 27% of cases SATzilla chooses its best component solver [XHHLB07].

While both SATzilla and ArgoSmArT adjust the solving process to the input formula, there are important differences between these two approaches. First, SATzilla is the system that chooses among its component solvers (7 solvers were used at the *SAT competition* 2007), and these solvers are used as they are. On the other hand, our approach aims at boosting performance of just a single, arbitrary, base solver by selecting strategies appropriate for an input formula (in the current setup it chooses between 21 strategies that happen to be the best for some of the families from the training corpus). Therefore these two approaches can be considered complementary. The advantage of the SATzilla approach, compared to the ArgoSmArT approach, is that it offers

an estimate of the running time. On the other hand, an important advantage of ArgoSmArT is that it can detect a family that the input formula belongs to.

Stochastic optimization of SAT solver parameters is described in [HBHH07]. It could be used for finding better strategies within our approach.

6 Conclusions and Future Work

We proposed a methodology for instance-based selection of solving strategies that can be applied to an arbitrary SAT solver which supports multiple solving strategies. We showed that the family a formula belongs to can be automatically recognized and that precision achieved was excellent. Also, we demonstrated that for each family of formulae, among many possible strategies, just a small number of strategies is appropriate for its solving. Along with significant correlation between syntactical similarities of formulae and similarities of strategies most appropriate for their solving, these conclusions form a firm basis for our strategy selection methodology.

The methodology was evaluated on two representative corpora. As for the overall performance, on the SAT2002 corpus a greater number of formulae was solved and the median time dropped more than 50%. The performance improvement was also demonstrated on a corpus different from the one the system was trained on. Overall, the results obtained are very good and show that the methodology is practically applicable and that further research in this, still new field, is feasible. We are planning to work on additional statistical analyzes of gathered data in order to gain a deeper insight into the nature of our best strategies and relationships between their component policies. We plan to further improve our system by using the stochastic parameter optimization, which would significantly decrease duration of the training phase. Also, we will try to combine our approach with the SATzilla approach by training SATzilla to choose between different strategies of a solver. While inspecting our data we came across the incompatibility of the MiniSAT forgetting strategy we used with the fast restarting strategies when forgetting quickly ceases. Impacts of these incompatibilities should be investigated and potentially better results achieved by changing the forget strategy. Also, a deeper analysis of behaviour of best strategies for k-SAT instances is planned.

Acknowledgements. We thank Mathematical Institute of Serbian Academy of Sciences and Arts for providing us access to their cluster computer.

References

[Bie08] Biere, A.: PicoSAT Essentials. Journal on Satisfiability, Boolean Modeling, and Computation (2008)

[Coo71] Cook, S.A.: The Complexity of Theorem-Proving Procedures. In: STOC 1971: Proceedings of the Third Annual ACM Symposium on Theory of Computing. ACM Press, New York (1971)

[DLL62] Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-Proving. Commun. ACM (1962)

[DP60] Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. J. ACM (1960)

[ES04] Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Theory and Applications of Satisfiability Testing (2004)

[GKSS07] Gomes, P., Kautz, H., Sabharwal, A., Selman, B.: Satisfiability Solvers. In: Handbook of Knowledge Representation, Elsevier, Amsterdam (2007)

[HBHH07] Hutter, F., Babic, D., Hoos, H.H., Hu, A.J.: Boosting Verification by Automatic Tuning of Decision Procedures. In: FMCAD 2007: Proceedings of the Formal Methods in Computer Aided Design, IEEE Computer Society Press, Los Alamitos (2007)

[KG07] Krstić, S., Goel, A.: Architecting solvers for SAT modulo theories: Nelson-oppen with DPLL. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS, vol. 4720, pp. 1–27. Springer, Heidelberg (2007)

[Kib07] Kibria, R.H.: Evolving a Neural Net-Based Decision and Search Heuristic for DPLL SAT Solvers. In: IJCNN (2007)

[LL98] Lobjois, L., Lemaitre, M.: Branch and Bound Algorithm Selection by Performance Prediction. In: AAAI, AAAI Press, Menlo Park (1998)

[LSZ93] Luby, M., Sinclair, A., Zuckerman, D.: Optimal Speedup of Las Vegas algorithms. Information Processing Letters (1993)

[Mar08] Marić, F.: Formalization and Implementation of SAT Solvers. Journal of Automated Reasoning (submitted, 2008)

[Mar09] Marić, F.: Flexible Implementation of SAT solvers. In: SAT 2009, (submitted, 2009)

[MMZ+01] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001 (2001)

[NBH+04] Nudelman, E., Leyton-Brown, K., Hoos, H.H., Devkar, A., Shoham, Y.: Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 438–452. Springer, Heidelberg (2004)

[NOT06] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). J. of the ACM (2006)

[PD07] Pipatsrisawat, K., Darwiche, A.: A Lightweight Component Caching Scheme for Satisfiability Solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)

[SM07] Samulowitz, H., Memisevic, R.: Learning to Solve QBF. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, AAAI Press, Menlo Park (2007)

[SM08] Smith-Miles, K.: Cross-Disciplinary Perspectives on Meta-Learning for Algorithm Selection. ACM Comput. Surv. (2008)

[TJK06] Tomovic, A., Janicic, P., Keselj, V.: n-Gram-Based Classification and Unsupervised Hierarchical Clustering of Genome Sequences. Computer Methods and Programs in Biomedicine (2006)

[XHHLB07] Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: The Design and Analysis of an Algorithm Portfolio for SAT. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 712–727. Springer, Heidelberg (2007)

[XHHLB08] Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based Algorithm Selection for SAT. Journal of Artificial Intelligence Research (2008)

[XSS09] Xu, Y., Stern, D., Samulowitz, H.: Learning Adaptation to Solve Constraint Satisfaction Problems. In: LION 3 (2009)

[ZMMM01] Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In: International Conference on Computer Aided Design (ICCAD) (2001)

Width-Based Restart Policies for Clause-Learning Satisfiability Solvers

Knot Pipatsrisawat and Adnan Darwiche

Computer Science Department
University of California, Los Angeles, USA
{thammakn,darwiche}@cs.ucla.edu

Abstract. In this paper, we present a new class of restart policies, called *width-based* policies, for modern clause-learning SAT solvers. The new policies encourage the solvers to find refutation proofs with small widths by determining restarting points based on the sizes of conflict clauses learned rather than the number of conflicts experienced by the solvers. We show that width-based restart policies can outperform traditional restart policies on some special classes of SAT problems. We then propose different ways of adjusting the width parameter of the policies. Our experiment on industrial problems shows that width-based policies are competitive with the restart policy used by many state-of-the-art solvers. Moreover, we find that the combination of these two types of restart policies yields improvements on many classes of problems.

1 Introduction

Restarting has become an essential component of modern SAT solvers since the work of Gomes *et al* [1], which pointed out a problem of backtracking algorithms on combinatorial problems. In the past, restart policies used by SAT solvers were mostly static and were based on the number of conflicts experienced by the solvers (e.g., [2,3,4,5]). The intuition behind these approaches is that conflicts indicate bad assignments. So, if the solver experiences a lot of conflicts, it might have made some bad assignments early on and restarting, together with a dynamic decision heuristic, may allow these assignments to be fixed. However, this class of approaches does not take into account the actual search behavior of the solvers and may yield a bad performance on even some easy problems.

Recently, some researchers tried to improve this idea further by studying dynamic restart policies. For example, in [6], the notion of *agility*, which approximates the diversity of the assignments recently explored by the solver, was used to prevent the solver from restarting too frequently. In [7], the authors argued that restarts should be triggered based on the number of conflicts experienced under each search branch and proposed some restart policies based on this idea.

It is well-known that modern clause-learning SAT solvers can be viewed as resolution engines, which produce refutation proofs on unsatisfiable problems [8]. From this perspective, the class of restart policies based on the number of conflicts can be viewed as a way of biasing the solvers to find short unsatisfiable proofs for unsatisfiable problems.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 341–355, 2009.

In this work, we utilize the notion of *proof width* [9], which can also be used to measure the quality of resolution proofs, to control restarts in clause-learning SAT solvers. In particular, we propose a new class of restart policies, called *width-based policies*. According to these policies, the solvers maintain a width limit at any moment and restart as soon as too many clauses of size greater than the limit are learned. This class of policies simply tries to encourage the solvers to find a refutation proof with a small width, which could also lead to a small proof. We demonstrate how simple width-based restart policies with constant width limits can significantly outperform policies used by state-of-the-art solvers on some families of SAT problems. Then, we propose a general algorithm for adjusting the width limits used in such policies. Finally, we evaluate several width-based policies based on this algorithm on various classes of problems.

The rest of this paper is organized as follows. We review some basic notations about resolution proofs and modern clause-learning SAT solvers in the next section. In Section 3, we review existing restart policies used by leading SAT solvers and describe width-based restart policies. In Section 4, we present the results of our empirical studies on some special classes of problems, which demonstrate the strengths of width-based policies. In Section 5, we describe a general algorithm for adjusting the width limit in width-based policies. Then, we present experimental results on industrial and crafted problems in Section 6. Finally, we discuss some related work in Section 7 and conclude in Section 8.

2 Preliminaries

In this section, we discuss some basic notions that form the basis of our later discussions. First, we review basic notations about resolution and resolution proofs. Then, we briefly describe how modern clause-learning SAT solvers work from a resolution perspective. Finally, we point out the relationship between our work and an existing SAT algorithm based on proof width.

2.1 Resolution Proofs

A *resolution* between two clauses $C_1 = (x \vee \alpha)$ and $C_2 = (\neg x \vee \beta)$ is the derivation of the clause $C = (\alpha \vee \beta)$. In this case, C is called the *resolvent* of the resolution. A resolution proof Π of clause C_k from CNF Δ is a sequence of clauses $\Pi = C_1, C_2, ..., C_k$, where each clause C_i is either in Δ or is the resolvent of some clauses preceding C_i. The *size* of Π is simply the number of clauses in it, while its *width* is the size of the largest clause in it. In this work, we are mostly interested in *refutation proofs*, which are resolution proofs of the empty clause (i.e., false) from unsatisfiable CNFs. The width of an unsatisfiable CNF is simply the smallest width of any of its refutation proofs.

2.2 Modern Clause-Learning SAT Solvers

A typical modern clause-learning SAT solver works by repeatedly making decisions and using unit resolution to derive implications. Upon a conflict, the solver

derives a conflict clause to allow unit resolution to see an implication that was missed earlier. Then, it backtracks, asserts the learned clause, and continues making decisions. This process is repeated until either a solution is found or the empty clause is derived. For a more detailed description see [10].

For example, consider the following CNF:

$$\begin{aligned}\Delta = &(\neg a \vee \neg b \vee c), (\neg a \vee \neg c \vee d), (\neg a \vee \neg c \vee e), (\neg a \vee \neg d \vee \neg e),\\ &(\neg a \vee c \vee d), (\neg a \vee c \vee e), (a \vee \neg b \vee c), (a \vee \neg b \vee \neg c),\\ &(a \vee b \vee \neg c), (a \vee b \vee e), (a \vee b \vee \neg f), (c \vee \neg e \vee f).\end{aligned}$$

We can view the execution of a clause-learning SAT solver as a series of decision making and clause learning. Table 1 shows the sequence of decisions made and clauses learned by the solver in chronological order. In this example, we assume that the solver makes decisions in alphabetical order and always sets decision variables to true. Implications derived by unit resolution after each decision and after each clause learning are also shown. Lastly, each step is associated with a level, which is simply the number of decisions currently in effect. After the first decision ($a = \mathsf{true}$), no implication is derived. However, after setting $b = \mathsf{true}$, unit resolution will derive implications c, d, e and find that $(\neg a \vee \neg d \vee \neg e)$ is falsified (indicated by false in the implication row). From this conflict, the solver will learn $(\neg a \vee \neg c)$ and backtrack to level 1. Applying unit resolution on this clause will result in implications $\neg c, d, e$ and another conflict, from which the solver learns $(\neg a)$ and backtracks to the top level (level 0). Asserting $(\neg a)$ produces only one implication. The next decision is $b = \mathsf{true}$, because a is already set to false. The solver will encounter yet another conflict and derive $(a \vee \neg b)$. Asserting this clause at the top level yields a conflict and (a) can be derived. Since the solver has learned $(\neg a)$ and (a), the empty clause (false) can be derived and the solver can now conclude that Δ is unsatisfiable.

Each conflict clause learned by the solver can be derived by resolving clauses present in the knowledge base of the solvers at the time of the conflict. Hence, when a clause-learning solver solves an unsatisfiable CNF, the conflict clauses learned by the solver can be thought of as traces of the refutation proof produced by the solver. A full refutation proof can be extracted from any run of clause-learning SAT solvers (on an unsatisfiable CNF) if the solvers keep track of every resolution performed during their executions [11].

Figure 1 shows the refutation proof of Δ produced by the solver in the above example, demonstrating how the conflict clauses come together to form a proof of the empty clause. The conflict clauses are enclosed in boxes in this figure. Other clauses in the proof are either original clauses in Δ or are intermediate

Table 1. An execution trace of a typical modern clause-learning SAT solver

Decisions/learned clauses	a	b	$(\neg a \vee \neg c)$	$(\neg a)$	b	$(a \vee \neg b)$	(a)
Implications	-	$c, d, e,$ false	$\neg c, d, e,$ false	$\neg a$	$c,$ false	$\neg c, e, \neg f,$ false	false
Levels	1	2	1	0	1	0	0

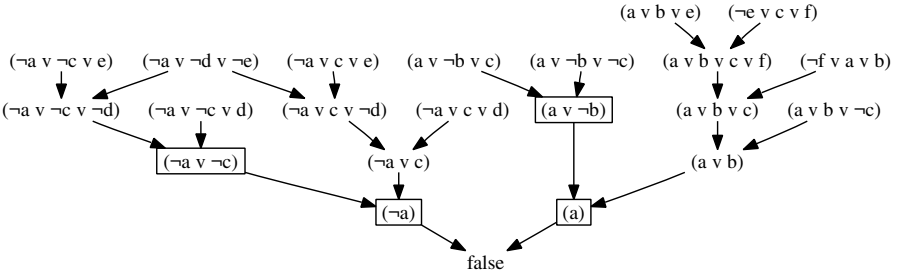


Fig. 1. A refutation proof generated by a modern SAT solver. Conflict clauses are shown in boxes.

resolvents, which are not kept by the solver. The width of this refutation proof is 4, because the longest clause, $(a \vee b \vee c \vee f)$, contains 4 literals.

2.3 A Width-Based Algorithm for SAT

Galil [12] proposed a SAT algorithm which runs in time exponential in the width of the CNF formula. This algorithm, which was later reformulated in [13] and [9], works by deriving all resolvents of size $\leq k$, for increasing k. Since there are only $O(n^k)$ clauses of size $\leq k$, where n is the total number of variables, this algorithm works well on formulas with bounded or small widths. Moreover, it was shown in [9] that this algorithm runs in time that is at most quasi-polynomial in the size of the smallest tree-like refutation proof (i.e., optimal DPLL).

Nevertheless, one drawback which limits the practicality of this approach is the amount of memory it requires. Even though the space complexity of the algorithm is only exponential in the width of the proof, in practice, this could be a serious limiting factor–especially when compared to the clause-learning descendants of DPLL, which perform resolution in a more directed way and keep only a fraction of the resolvents in the knowledge base.

The restart policies that we propose in this work can be thought of as a way to efficiently combine the benefits of both approaches. In other words, our approach can be viewed as a way of using the low memory requirement of modern clause-learning SAT algorithms to loosely imitate the above width-based algorithm.

3 Existing and Width-Based Restart Policies

3.1 Existing Restart Policies

In this section, we briefly review existing restart policies used by state-of-the-art clause-learning SAT solvers. One common characteristic of these policies is that they use the number of conflicts experienced by the solver to determine restarting points. According to these policies, the solvers restart as soon as the number of conflicts (since the last restart) exceeds the current threshold. Since a typical clause-learning SAT solver learns one clause per conflict, this class of restart

policies can be viewed as a way of roughly enforcing a limit on the size of the refutation proof currently considered by the solver. For this reason, we will refer to this class of policies as *size-based restart policies*. These policies only differ in the way the size threshold is updated at each restart. In the following discussion, we group these policies based on their methods of updating the threshold.

1. Arithmetic series: the threshold is increased by a constant amount (≥ 0) at every restart. This type of policy was used (with different parameters) in zChaff (2004) [14], Berkmin [4], Siege [10], and Eureka [15].
2. Geometric series: the threshold is multiplied by a constant factor (> 1) at every restart. This type of policy is used in MiniSat 1.14 and 2.0 [3].
3. Inner-outer geometric series: the solver maintains two thresholds (inner and outer). The inner threshold is used to trigger restarts and is multiplied by a constant factor (> 1) at every restart. However, if the value of the inner threshold exceeds the value of the outer threshold, the inner threshold is reset back to its minimum value, while the outer threshold is multiplied by a constant factor (> 1). PicoSAT [16] uses this policy.
4. Luby's series [5]: the threshold is updated according to the following sequence: $x, x, 2x, x, x, 2x, 4x, x, x, 2x, x, x, 2x, 4x, 8x, ...$, where x is a constant called Luby's unit (see [5] for more details). TiniSAT [17] , Rsat [18], and the latest version of MiniSat [19] use this restart policy.

Clearly, one drawback of these policies is that they are insensitive to the actual search behavior of the solver. Dynamic policies leverage on additional information generated during the execution of the solver to improve performance. In [6], the diversity of partial assignments current explored by the solver is used to create another layer of control, which helps prevent the solvers from restarting too frequently on some problems. In [7], the number of conflicts experienced below each search branch is used to determine when to restart.

3.2 Width-Based Restart Policy

Our approach to restart is based on a different model, which does not rely on the number of conflicts experienced by the solvers. Rather, we pay attention to the sizes of conflict clauses learned by the solver. If the CNF in question has a short refutation proof, the well-known result in [9] states that the formula must also have a refutation proof with a small width. If a formula has a proof with width k, then we know we can certainly find such a proof in time $O(n^k)$. Thus, enforcing a limit on proof width allows us to bound not just the size of the proof found, but also the amount of work needed to find such a proof.

In a *width-based* restart policy, the solver maintains a width limit W at any given moment. Any conflict clause whose length is greater than the current value of W is called a *violating clause*. In the most general form, the solver restarts as soon as it derives N or more violating clauses since the last time it restarted. For example, if $W = 10$ and $N = 3$, the solver will restart once at least 3 clauses

of length 11 or greater are derived.[1] Note that violating clauses are not deleted by the solver immediately, but are treated normally just like non-violating ones. During the execution of the solver, the value of W may be kept constant or changed based on some criteria. This choice does not affect the completeness of clause-learning solvers as long as a complete clause-deleting policy is employed. Note also that, the absence of conflict clauses of size $> W$ does not guarantee that the width of the refutation proof generated by the solver will be $\leq W$. For instance, consider again the refutation proof in Figure 1. Even though every conflict clause in this proof has length at most 2, the width of this proof is actually 4. In general, the clause-learning algorithm may generate some long intermediate clauses, which do not get learned by the solver. These clauses are not taken into account in our approach.

4 Potential Benefits of Width-Based Policies

In this section, we demonstrate the potential benefits of width-based restart policies by comparing them against size-based policies on interesting SAT problems with relatively small widths. If the width k of an unsatisfiable CNF is given, one natural restart policy is to restart as soon as a conflict clause of size $> k$ is learned. To demonstrate the benefits of this approach, we will show that a width-based policy with an appropriate width limit can significantly outperform size-based policies used by state-of-the-art solvers. All experiments discussed in this section were performed on a computer with a 1.83GHz CPU and 1.5GB RAM. We set the timeout limit to 2000 seconds. We used Rsat [18] (without the preprocessor) in the following experiments.

In the first experiment, we used the unsatisfiable grid pebbling formulas with two variables per node as described in [8]. All formulas have a very small constant width (4). Nevertheless, this family was shown to be difficult for tree-like resolution [20]. Evaluated in this experiment are size-based policies (using arithmetic, geometric, and Luby's series), and a width-based restart policy. An increment of 700 was used for the arithmetic series (like zChaff 2004), a factor of 1.5 was used for the geometric series (like MiniSAT 1.14), and the Luby's unit was set to 512 (like TiniSAT, Rsat) for the Luby's series. The width limit of the width-based policy was set to 4. Table 2 reports the running time of Rsat with the considered policies on this set of problems. The first row shows the grid sizes of the grid pebbling formulas (i.e., the numbers of layers in [8]). Each remaining row shows the running time of a restart policy on these problems.

Figure 2 (a) is a plot of the running time of all restart policies as functions of grid size. According to the result, the geometric size-based policy has the worst performance as it begins to timeout when the grid size is only about 60. The policy based on Luby's series starts to timeout when the grid size gets larger than 220. The policy based on arithmetic series performs quite well on these problems

[1] We found that restarting only when the solver is not in a conflict state simplifies the implementation. In this approach, it is possible for the number of violating clauses to be (usually slightly) greater than N when the solver actually restarts.

Table 2. Running time (in seconds) of Rsat with different restart policies on unsatisfiable grid pebbling formulas of different sizes

Grid size	51	52	53	151	152	153	201	202	203	238	239	240
Arith.	1	1	3	124	111	81	193	225	262	410	477	T/O
Geo.	21	210	379	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O
Luby	3	3	4	200	87	202	569	903	640	T/O	T/O	T/O
Width	1	1	1	36	27	21	85	93	108	244	414	257

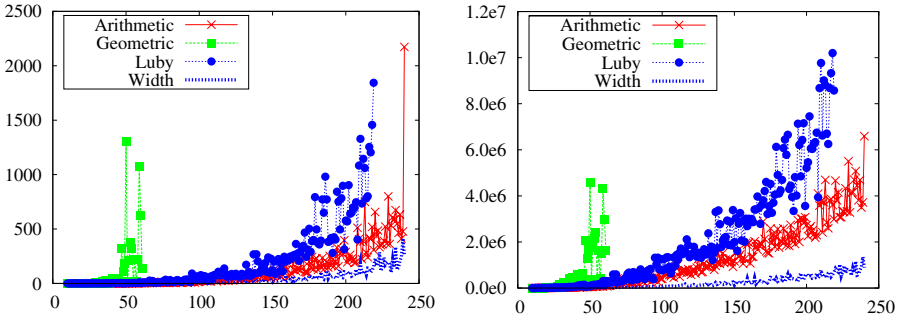


Fig. 2. Performance of Rsat with different restart policies on grid pebbling problems. In both plots, the x-axes represents the grid size. The y-axis of left (right) plot represents the running time (number of conflicts).

and could solve the problems up to grid size equal to 239, because it has relatively short restart periods, which are very effective for preventing the solver from getting stuck deriving long, useless clauses. In any case, the width-based policy has the best performance on these problems–up to an order of magnitude faster than the Luby size-based policy and several times faster than the arithmetic size-based policy. The superiority of the width-based policy becomes even more apparent when the solvers are compared in terms of number of conflicts needed to solve the problems (Figure 2 (b)). In our experiment (result not shown here), Rsat with the width-based restart policy could solve the problem with grid size equal 500 in 1,373 seconds.

We also experimented with the satisfiable version of the grid pebbling formulas (as described in [8]). Again, the width-based policy dominates the size-based policies both in terms of running time and conflicts. For instance, at grid size equal 260, the arithmetic size-based policy took 421 seconds, the Luby size-based policy took 922 seconds, while the width-based policy only took 134 seconds (the geometric size-based timed out for grid size ≥ 150).

Figure 3 shows similar results for the GT_n family of unsatisfiable problems [8]. A GT_n formula is a formula over $\sim n^2$ variables whose width is linear in n. In this case, we use the same set of size-based policies and set the width limit of the width-based policy to be 20. The result shows that the geometric size-based policy timed out for $n \geq 21$, the arithmetic size-based policy timed out after

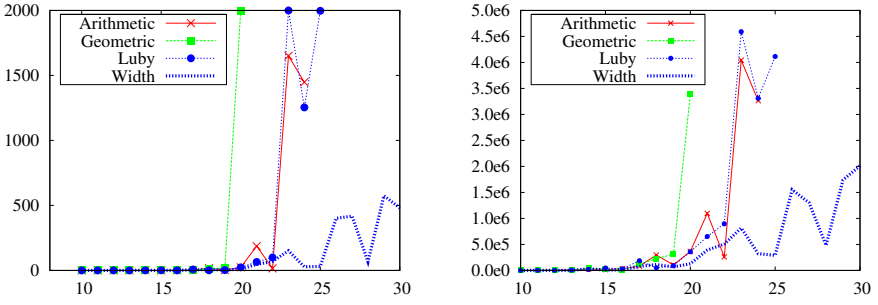


Fig. 3. (Left) running time of Rsat with different policies on unsatisfiable GT_n formulas as functions of n. (Right) number of conflicts as functions of n.

$n = 24$, and Luby size-based policy timed out after $n = 25$. The width-based policy was able to solve the problem with $n = 30$ in about 480 seconds. To give a sense of the hardness of these problems, consider the problem gt-ordering-sat-gt-040.sat05-1297.reshuffled-07 from the crafted category of the SAT competition 2007. In the competition, this problem was not solved by MiniSat, Rsat, TiniSat, or PicoSat under a 5000-second timeout. However, it could be solved with a width-based policy with the width limit set to 20 in 40 seconds.

We found that some industrial problems can also be solved without requiring any long clauses to be learned. For example, consider the dspam_dump family from the SAT competition 2007. These problems were generated by CALYSTO [21,22] from a software verification task (NULL-pointer dereferencing) on a spam filter. Based on our experiment, these unsatisfiable problems could be easily solved without any long conflict clauses, yet clause-learning solvers may derive many long clauses. Table 3 shows the information collected from running Rsat, which uses Luby size-based policy, on selected problems from this family.[2] The first column shows the names of the problems. The second and third columns reports the number of variables and clauses. The forth column reports the size of the largest conflict clause of each refutation proof found by Rsat. The remaining columns show the running time, the size of the largest conflict clause learned, and the percentage of conflict clauses longer than the ones needed by the proof, respectively. Rsat with Luby size-based restart policy took over 2,200 seconds to solve all four problems. Moreover, most of the clauses learned by the solver were unnecessarily long. Nevertheless, these problems become easy if a width-based restart policy (with an appropriate width limit) is used. Rsat using a width-based restart policy with width limit set to 6 can solve all these problems within 25 seconds (combined).

The results of these experiments show that a width-based restart policy (with the right width limit) can dramatically reduce the running time of the solver on some problems. However, in practice, the width of the problem is not known

[2] We disabled conflict clause deletion in order to collect some statistics.

Table 3. Information obtained from the executions of Rsat (using Luby size-based restart policy) on dspam_dump problems from SAT competition 2007

Problem	vars	clauses	largest clause in proof	running time (s)	largest clause size	% long clauses
dspam_dump_vc1080	118,298	372,017	4	205	1,147	86.5
dspam_dump_vc1081	118,426	372,337	3	1,716	2,533	96.6
dspam_dump_vc1103	280,972	921,211	5	195	1,408	85.9
dspam_dump_vc1104	280,972	921,147	3	169	1,229	89.1

beforehand and width computation is believed to be very hard [23].[3] Therefore, we need to adjust the width limit dynamically to obtain good performance.

5 Adjusting Width Limits

In this section, we describe a general algorithm for updating the width limit and propose several methods for updating the width limit. In a general width-based policy, at any given time, the solver maintains one width limit W and restarts as soon as it derives N or more conflict clauses with size greater than W. We consider the following methods for updating W.

1. Arithmetic series: after R restarts at the current limit, W is incremented by a constant C_1.
2. Geometric series: after R restarts at the current limit, W is multiplied by a constant C_2.
3. Inner-outer geometric series: after R restarts at the current limit, W is multiplied by a constant factor C_2. However, as soon as the value of W reaches V, it is reset to its initial value and V is multiplied by a constant factor C_3.
4. Luby series: after R restarts at the current limit, W is updated to be the next number in the Luby series with unit U.

The values of $N, R, U, V, C_1, C_2, C_3$ are parameters that need to be fine-tuned for these policies. Nevertheless, finding optimal values for these parameters is not the main focus of this work. In subsequent experiments, we set $W = 15$ (initially), $N = 10, R = 1, U = 6, V = 20$ (initially), $C_1 = 1, C_2 = 1.005, C_3 = 1.05$. One can certainly envision policies which adjust these parameters dynamically.

In addition to these pure width-based restart policies, we also consider their combinations with a size-based restart policy. In this case, the proofs explored by the solvers are loosely bounded both in terms of size and width. For this combination, we used the size-based restart policy based on Luby's series, which has been found to yield good performance on industrial problems [5]. In such a hybrid policy, the width limit and the size threshold are enforced independently. That is, the solver restarts based on clause size as described above and, moreover, if the number of conflicts experienced by the solver (since the last time it restarted based on number of conflicts) reaches the size threshold, the solver also restarts (without updating the width limit).

[3] The problem of computing width was conjectured to be EXPTIME-complete.

6 Experimental Results

In this section, we evaluate the performance of the restart policies discussed in the previous section. All experiments were performed on a computer with a 3.8GHz CPU and 4GB of RAM. The timeout was set to 30 minutes per problem. The use of proprocessor was diabled in all experiments in order to obtain the impacts of the restart policies on pure clause-learning solvers.

In the first experiment, we compared the proposed width-based restart policies against the Luby size-based policy (unit=512) on 175 industrial problems from the last SAT competition.[4] Table 4 reports the number of problems solved by each policy. According to the table, (pure) width-based policies seem to consistently result in worse performance on satisfiable problems. This could be due to the significant increase in the number of restarts introduced by the width-based policies. More frequent restarts cause the solver to remake many decisions and spending more time in unit propagation.[5] For unsatisfiable problems, the results are more comparable. The geometric width-based policy actually solved 5 more unsatisfiable problems than the size-based policy. Overall, the performance of the geometric width-based policy is about the same as that of the Luby size-based policy. This result demonstrates that width-based policies, could be competitive to a size-based policy. Note that the parameters used in our experiment were not fine-tuned. The table also shows that the hybrid policies consistently outperformed the Luby size-based policy (except the one with inner-outer width-based policy, which performed poorly on satisfiable problems). The combination of the geometric width-based policy and the Luby size-based policy, in particular, appears to be the best version on this set of problems.

Table 4. Number of industrial problems from the SAT competition 2007 solved by different restart policies

Policy	Solved problems		
	Total	SAT	UNSAT
Size-based (Luby,unit=512)	107	49	58
Width-based (Arith.)	100	46	54
Width-based (Geo.)	108	45	63
Width-based (Luby)	103	47	56
Width-based (In-out.)	90	36	54
Width-based (Arith.)+size-based (Luby)	110	48	62
Width-based (Geo.)+size-based (Luby)	115	52	63
Width-based (Luby)+size-based (Luby)	114	54	60
Width-based (In-out.)+size-based (Luby)	106	43	63

[4] Obtained from http://www.satcompetition.org.

[5] For example, whenever the width-based policy (geo.) solves the problem with the number of conflicts comparable (within 5%) to that of the size-based policy (Luby), it makes 42% more decisions on average.

Table 5. Number of problems solved by different versions of Rsat

Family	Total	Solved problems					
		Rsat	Rsat-ag	Rsat-lc	Rsat-ws-1	Rsat-ws-2	Rsat-ws-3
SAT comp. 07	175	107	109	114	**115**	114	110
SAT-Race'06	100	86	87	84	**90**	88	84
dlx-iq-unsat-1.0	32	11	12	7	**18**	14	17
fvp-unsat-1.0,2.0,3.0	32	26	26	26	26	26	26
liveness-sat-1.0	10	5	5	6	**7**	6	6
liveness-unsat-2.0	9	3	3	3	3	3	3
pipe-ooo-1.0,1.1	29	12	12	11	**13**	12	**13**
pipe-unsat-1.0,1.1	27	14	14	13	**16**	**16**	**16**
vliw-unsat-2.0,4.0	13	0	0	0	**2**	0	0
Total	427	264	268	264	**290**	279	275

Next, we compare the best policies from the previous experiment against other dynamic restart policies in order to establish a context for the benefit of width-based restart policies. This time, we also consider problems from SAT-Race 2006 and some hardware verification problems.[6] Considered in this experiment are the following versions of Rsat.

1. Rsat 2.00 (SAT competition 2007 version) [Rsat]. This version of Rsat uses a size-based restart policy based on Luby's series with Luby's unit set to 512.
2. Rsat with agility-based restart policy [Rsat-ag]. Restart is disabled if the agility of the solver is greater than 0.25 (as described in [6]).
3. Rsat with local restarts [Rsat-lc]. A restart is triggered only when the number of conflicts under some search branch exceeds the threshold (as described in [7]). The Luby's series (unit=512) is used to update the threshold.
4. Rsat with hybrid restart policies [Rsat-ws-1,2,3]. The Luby size-based policy is combined with (1) the geometric width-based, (2) the Luby width-based, and (3) the arithmetic width-based policies.

Table 5 shows the number of problems solved by each solver. Overall, the agility-based restart policy allows Rsat to solve a few more problems, while local restarts yield about the same performance as the original Rsat. These techniques seem to be most effective on problems from the SAT competition and SAT-Race'06.[7] The hybrid restart policies have the best overall performance. Rsat-ws-1,2,3 solved 26, 15, 11 more problems than Rsat, respectively. Clearly, the geometric width-based and Luby size-based combination (Rsat-ws-1) yielded the best performance. Note that, in a hybrid policy, number of restarts triggered by width violations usually dominates size-based restarts.[8] Moreover, using width-based restart policies also

[6] The hardware verification problems were obtained from http://www.miroslav-velev.com/sat_benchmarks.html

[7] We did not optimize the parameters used in these techniques.

[8] E.g., on SAT'07 problems, 78% of restarts are width-based, while on SAT-Race'06 problems, 73% of restarts are width-based (based on the execution of Rsat-ws-1).

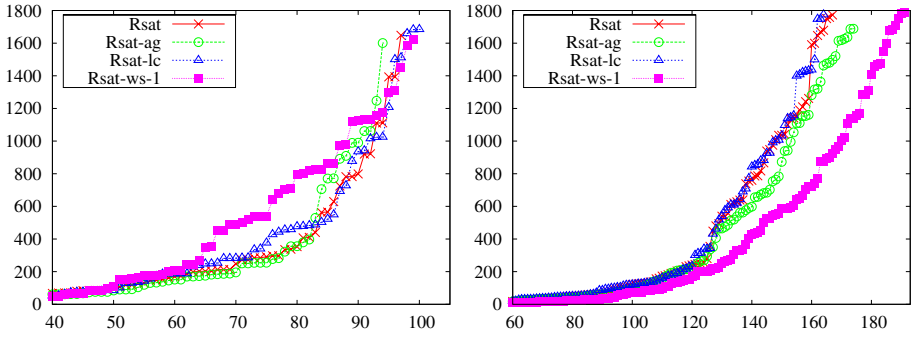


Fig. 4. Running time profiles of Rsat with different restart policies on (left) satisfiable and (right) unsatisfiable problems. Both x-axes represent the number of solved problems, while the y-axes represent running time in seconds.

tends to reduce the sizes of clauses learned. On the SAT competition 2007 problems, the sizes of clauses learned by Rsat-ws-1 is only 76% of those learned by Rsat on average. On SAT-Race'06 problems, this percentage is 81%.

Figure 4 shows the running time profiles of different versions of Rsat on the sets of problems in Table 5. For clarity, we show only one profile of Rsat with a hybrid policy, Rsat-ws-1. The left plot shows the profiles on satisfiable problems, while the right plot shows the profiles on unsatisfiable problems. The left plot indicates that Rsat-ws-1 actually performed slightly worse than Rsat on satisfiable problems, even though it ended up solving 2 more problems. The right plot, however, shows that Rsat-ws-1 is the clear winner on unsatisfiable problems. Rsat-ag and Rsat-lc appear to have comparable profiles to Rsat.

Our experiment on crafted problems from the SAT competition 2007 also confirms the benefit of the hybrid policy. We compared the performance of Rsat against the best hybrid policy, Rsat-ws-1. Figure 5 shows the running time profiles of these solvers on satisfiable (left) and unsatisfiable (right) problems. Even though Rsat-ws-1 solved fewer satisfiable problems, it took less time on most of the ones it solved. Moreover, Rsat-ws-1 solved 7 more unsatisfiable problems and took less time on those that both versions could solve.

We also tested the hybrid restart policy on MiniSat 2.0 (no preprocessor). By default, MiniSat uses a geometric size-based restart policy. We added the geometric width-based policy on top of this to obtain a hybrid policy (geometric width-based + geometric size-based). Our experiment on the industrial problems of SAT competition 2007 showed that MiniSat solved 109 problems, while MiniSat with the hybrid restart policy solved 114 problems.[9]

[9] Here, we used progress saving [24] in both versions of MiniSat as this technique seems to allow a frequent restart policy to realize its full potential. Without progress saving, both versions solved fewer problems and the improvement is less significant.

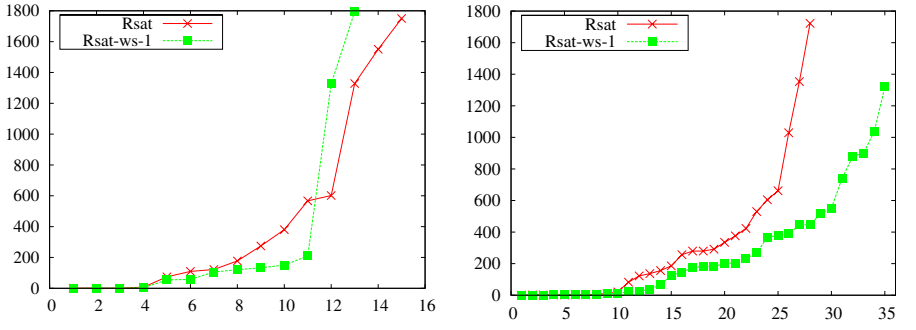


Fig. 5. Running time profiles of Rsat with different restart policies on (left) satisfiable and (right) unsatisfiable problems from the crafted category of SAT'07 competition

7 Related Work

The concept of *space-bounded learning* has long been studied in CSP [25,26] and SAT [27]. This approach restricts the algorithm to learning only those constraints with a limited number of variables. The restart policies we propose still allows long clauses to be learned, but use restarts to discourage their learning.

A technique in SAT, which tries to achieve similar goals, is known as *decision stack shrinking*, which was introduced by JeruSAT [28] and later used by zChaff2004 [14]. This technique tries to force the solver to discover a conflict at a lower level, thus deriving shorter a conflict clause. The technique is invoked whenever a long conflict clause is learned. Upon learning such a clause, the solver examines the decision levels of the literals in the clause and backtracks to the lowest level that is sufficiently smaller than the next higher level of any literal in the conflict clause. The solver then makes assignments in order to falsify the conflict clause (and run into the same conflict). Since some of the variables unassigned by the backtrack may not get assigned by the time the new conflict is discovered, the size of the decision stack is likely going to reduce, leading to potentially a shorter conflict clause. Our approach utilizes restarts to direct the solvers away from undesirable parts of the search space, thus inducing shorter conflict clauses. Decision stack shrinking, however, aims at improving the quality of conflict clause learned from a given (or similar) conflict.

8 Conclusions

We presented a new class of restart policies, called width-based restart policies, for clause-learning SAT solvers. These policies trigger a restart whenever the number of long clauses learned by the solver is sufficiently large. They can be thought of as ways to encourage the solvers to discover refutation proofs with small widths (instead of small sizes as done in traditional policies). Our study shows that width-based restart policies can be orders of magnitude faster than policies based on number of conflicts on special classes of problems. We then

propose a general algorithm for adjusting the width limits of width-based policies. Our experiment on industrial problems showed that pure width-based policies are competitive to the policy used by state-of-the-art solvers. Moreover, we show that width-based policies, when combined with a size-based policy, can lead to significant improvements on industrial and crafted problems.

References

1. Gomes, C.P., Selman, B., Crato, N.: Heavy-tailed distributions in combinatorial search. In: Principles and Practice of Constraint Programming, pp. 121–135 (1997)
2. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient sat solver. In: Proc. of DAC 2001, pp. 530–535 (2001)
3. Eén, N., Sörensson, N.: An extensible sat-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
4. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat-solver. In: DATE 2002, pp. 142–149 (2002)
5. Huang, J.: The effect of restarts on the efficiency of clause learning. In: Proc. of IJCAI 2007, pp. 2318–2323 (2007)
6. Biere, A.: Adaptive restart strategies for conflict driven sat solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 28–33. Springer, Heidelberg (2008)
7. Ryvchin, V., Strichman, O.: Local restarts. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 271–276. Springer, Heidelberg (2008)
8. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. JAIR 22, 319–351 (2004)
9. Ben-Sasson, E., Wigderson, A.: Short proofs are narrow—resolution made simple. J. ACM 48(2), 149–169 (2001)
10. Ryan, L.: Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, Simon Fraser University (2004)
11. Zhang, L., Malik, S.: Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE 2003, pp. 880–885 (2003)
12. Galil, Z.: On resolution with clauses of bounded size. SIAM Journal on Computing 6(3), 444–459 (1977)
13. Beame, P., Pitassi, T.: Simplified and improved resolution lower bounds. In: Annual IEEE Symposium on Foundations of Computer Science, p. 274 (1996)
14. Mahajan, Y.S., Fu, Z., Malik, S.: Zchaff2004: An efficient sat solver. In: Proc. of SAT 2005, pp. 360–375 (2005)
15. Nadel, A., Gordon, M., Patti, A., Hanna, Z.: Eureka-2006 sat solver Solver description for SAT-Race 2006 (2006)
16. Biere, A.: Picosat essentials. JSAT, 75–97 (2008)
17. Huang, J.: A case for simple SAT solvers. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 839–846. Springer, Heidelberg (2007)
18. Pipatsrisawat, K., Darwiche, A.: Rsat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA (2007)
19. Sörensson, N., Eén, N.: Minisat 2.1 and minisat++ 1.0–sat race 2008 editions (2008)

20. Ben-Sasson, E., Impagliazzo, R., Wigderson, A.: Near optimal separation of tree-like and general resolution. Combinatorica 24(4), 585–603 (2004)
21. Babić, D., Hu, A.J.: Structural Abstraction of Software Verification Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 371–383. Springer, Heidelberg (2007)
22. Babić, D., Hu, A.J.: Calysto: Scalable and Precise Extended Static Checking. In: Proc. of ICSE 2008, pp. 211–220 (2008)
23. Hertel, A., Urquhart, A.: Comments on eccc report tr06-133: The resolution width problem is exptime-complete. Technical Report TR09-003, ECCC (2009)
24. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
25. Dechter, R.: Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. Artif. Intell. 41(3), 273–312 (1990)
26. Bayardo, R.J., Miranker, D.P.: A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In: AAAI 1996, pp. 298–304 (1996)
27. Bayardo, R.J.J., Schrag, R.C.: Using CSP look-back techniques to solve real-world SAT instances. In: Proc. of AAAI 1997, Providence, Rhode Island, pp. 203–208 (1997)
28. Nadel, A.: Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, The Hebrew University (2002)

Problem-Sensitive Restart Heuristics for the DPLL Procedure*

Carsten Sinz and Markus Iser

Research Group "Verification meets Algorithm Engineering"
Institute for Theoretical Computer Science
University of Karlsruhe, Germany
{sinz,iser}@ira.uka.de

Abstract. Search restarts have shown great potential in speeding up SAT solvers based on the DPLL procedure. However, most restart policies presented so far do not take the problem structure into account. In this paper we present several new problem-sensitive restart heuristics. They all observe different search parameters like conflict level or backtrack level over time and, based on their development, decide whether to perform a restart or not. We also present a Java tool to visualize these search parameters on a given SAT instance over time in order to analyze existing heuristics and develop new one.

1 Introduction

Randomization and restart policies have been incorporated into SAT solvers since the mid 90s [1,2]. In a search procedure, a *restart* cancels the search for a solution after a certain number of steps, and then starts over again performing another solving attempt. To prevent the solver from generating the same search tree repeatedly, in early work on restarts randomization has been added to the decision heuristics of the DPLL procedure, thus modifying the search tree slightly on each run. With the advent of conflict driven clause learning (CDCL) SAT solvers [3,4], new ways to modify the search tree (to prevent repeated searches without progress) emerged, as these solvers dynamically compute *variable activities*, which can be taken into consideration after a restart: variables with highest activity (i.e. those occurring most frequently in recently learned clauses) are branched on first. Using learned clauses also allows to carry over results from previously cancelled attempts, as learned clauses need not be dropped after a restart. So the time spent in a failed search is not wasted. The theoretical groundwork for restarts was laid down in the seminal paper by Gomes *et al.* [6], where they explained the success of restarts by *heavy-tailed distributions*, which occur in randomized searches.

Current DPLL SAT solvers implement different restart heuristics or *restart schemes*. MiniSat 2.0, the latest publicly available version of the well-known MiniSat [7] solver, implements the RGR strategy (*randomization and geometric restarts*) proposed by Walsh [8], i.e. the n-th restart is performed $k \cdot \alpha^{n-1}$ steps after the previous restart (where $k = 100$ and $\alpha = 1.5$ by default, and steps refer to the number of conflicts).

* This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 356–362, 2009.

Other, more recent restart schemes are based on Luby sequences [9] or inner/outer restarts.

However, until recently, restart schemes for DPLL solvers have been static, in the sense that they are the same for each SAT instance, do not change during search, and thus are not *problem-sensitive*.[1] To the best of our knowledge, the first attempt to bring dynamic restart policies to DPLL solvers was made by Biere [12]. His adaptive restart strategy ANRFA (*average number of recently flipped assignments*) tries to estimate the *agility*of the on-going search process by taking the number of variable *flips* (determined with respect to the memorized phases) into account. A high number of flips indicates a high agility, which is considered good, whereas a low number of flips might indicate that the search process got stuck and a restart might be advantageous.

In this paper we present a set of new problem-sensitive restart policies. They are all based on observing a problem parameter during search and, depending on the development of this parameter, decide whether or not to perform a restart. We have also developed a Java tool for monitoring and visualizing such search parameters over time for a given SAT instance.

2 Problem-Sensitive Restart Heuristics

Accurately estimating the progress of a search process is a non-trivial task. However, some search parameters of a CDCL search—like length of learned clauses, search depth, or backtrack level—give hints whether the search is progressing fast or slowly. Based on these parameters, we have developed a set of new restart policies, which we will now describe in more detail. All our strategies are based on observing search parameters over time. Typically, we average over the last few parameter values that have occurred at a certain state during the search (in order to obtain more stable results). To put the most recent values of a search parameter into relation with "typical" values for the problem instance, we employ a short-term as well as a long-term memory for each parameter (storing the last L resp. S values for each). The short-term memory allows computing the current (smoothed) average value of the parameter, whereas the long-term memory is used to determine an average value over a longer period of time to compute, in a sense, a "typical" value for the parameter on this instance.

Such averaging over last (temporal) values is also know as *moving average* (or running average) in statistics. To compute the moving average of a parameter x, we use an array to store its $L+1$ most recent values $x_{n-L}, \ldots, x_n$. The same array can be used for both the long-term and short-term moving averages. It is used as a cyclic buffer with a pointer to the most recent entry. New entries move the pointer forward and overwrite the oldest value. Computing the moving average (A_n) over the last L values can be done efficiently (without summing over the whole array) in an iterative manner:

$$A_n = A_{n-1} + \frac{x_n - x_{n-L}}{L} \quad .$$

During the first L steps we just fill the array with values and, starting with step $L+1$, we use it to compute average values and to decide whether to restart or not. We record

[1] For *local search* solvers, problem-sensitive restart policies have been known for some time [10,11].

parameter values at each leaf of the search tree (i.e. on each conflict), such that the notion of step coincides with the number of conflicts. Parameters we found suitable for monitoring over time include

Conflict level: the height of the search tree when a conflict occurred.
Backtrack level: the height of the search tree to which the solver jumped back.
Length of learned clauses: the length of the currently learned clause.
Trail size: The total number of assigned variables when a conflict occurred (including variables assigned by unit propagation).

To determine whether we should perform a restart, we selected one of these parameters and tracked its evolution over time. It is possible to combine several parameters in a restart heuristic, but we have not made any experiments in this direction so far. We describe our restart heuristics exemplarily for the parameter **conflict level**.

Ratio of long term vs. short term average (R): This heuristic assumes that (relatively) low conflict levels are preferable to high conflict levels. The intuition is that uniformly low conflict levels span up a smaller search space. Moreover, the conflict clauses produced on lower conflict levels are potentially shorter and thus prune a larger fraction of the search space. Low conflict levels are determined in relation to the long-term average value: as soon as the short-term average conflict level (c_S) is much higher ($c_S/c_L \geq f_T$) than the long-term average conflict level (c_L), we perform a restart. Here f_T is a fixed threshold factor.

Avoidance of plateaus (P): When during search the same (high) conflict level occurs over and over again, this could indicate that the search got stuck. To avoid such a situation, we count the number S of minimal values of the conflict level over the last L steps. If the minimum occurs more often than a fixed number of times (given as a fraction of L) and the minimum is larger than a threshold value ($c_{T,\min}$), a restart is performed.

Preference for high variance (V): Similar to the last heuristic, and related to Biere's notion of *agility*, the intuition behind this restart scheme is to avoid situations where the conflict level is mainly constant (low variance). We compute the variance (resp. the standard deviation) over the last L conflict levels by $\sigma^2 = \frac{1}{L-1} \cdot \sum_{i=n-L+1}^{n} (x_i - \mu)^2$, where $\mu = \frac{1}{L} \cdot \sum_{i=n-L+1}^{n} x_i$ is the mean value of the last L conflict levels. If σ^2 is smaller than a threshold value $c_{T,\sigma}$, a restart is performed.

3 Experimental Evaluation

We have implemented the restart heuristics mentioned in the previous section on top of MiniSat 2.0.[2] The implementation required only minor modifications, including addition of the array storing recent versions of a search parameter.[3]

[2] http://minisat.se/downloads/minisat2-070721.zip

[3] In CDCL solvers, restarts are often closely tied to the deletion of learned clauses ("garbage collection"). In our extensions of MiniSat we left the clause deletion intervals unchanged. Also note that only either the restart intervals or the clause deletion intervals have to be gradually increased to ascertain completeness of the search procedure.

We have made experiments with all search parameters mentioned in the previous section, but will report only on results based on the **backtrack level** as, without further tuning of the heuristics' threshold values, further investigation of this parameter seemed most promising. For heuristic R, we have used a threshold factor of $f_T = 3.5$, and values $L = 30$ and $S = 7$ for long and short term ranges. For restart scheme P, we have determined best parameters for L and S by a sequence of experiments (using the *manolios* benchmark set), which resulted in $L = 24$ and $S = 4$ as optimal values. The restart threshold $c_{T,\min}$ was set to $100 \cdot \frac{V}{C}$, where V and C denote the number of variables resp. clauses of the instance. For heuristic V, we have set L to 30 and $c_{T,\sigma}$ to $1.3^2 = 1.69$. We compared our heuristics with both the original MiniSat 2.0 version[4] (denoted by M in the tables) and a restart heuristics that makes (frequent) restarts after a constant number of 200 steps (denoted by C in the tables).

Table 1. Comparison of problem-sensitive restart strategies

benchmark	# instances			# solved sat					# solved unsat					# solved total				
family	sat	unsat	total	P	R	V	C	M	P	R	V	C	M	P	R	V	C	M
manolios	0	210	210	–	–	–	–	–	**166**	145	161	158	151	**166**	145	161	158	151
velev-pipe	0	27	27	–	–	–	–	–	**16**	6	15	**16**	8	**16**	6	15	**16**	8
sat-race-2006	43	57	100	26	29	28	22	**36**	**50**	33	48	45	39	**76**	62	**76**	67	75
satcomp-07 indust.	68	107	175	36	37	**44**	40	37	**63**	51	55	57	53	**99**	88	**99**	97	90
satcomp-07 crafted	≥34	≥95	201	16	17	9	9	**22**	36	**48**	15	25	46	52	65	24	34	**68**
sat-race-2008	48	52	100	24	29	25	22	**33**	37	27	31	**41**	30	61	56	56	**63**	**63**

Table 1 shows the results of different restart heuristics on a number of benchmark families. All experiments were performed under SuSE Linux on machines equipped with an Intel Xeon E5430 processor running at 2.66 GHz and 16 GB of RAM. We set a time limit of 900 seconds per instance and solver for all of our experiments. The test set *manolios* is a parameterized benchmark suite consisting of hard pipelined-machine-verification problems. The *velev-pipe* family includes Velev's *pipe-unsat-1.0* and *pipe-unsat-1.1* problem sets.[5] The other benchmarks stem from previous SAT Competitions (2007, *industrial* and *crafted* category) and SAT-Races (2006 and 2008).[6] Table 1 shows the number of instances contained in each benchmark package, split into satisfiable and unsatisfiable instances. The following columns report on the number of instances that could be solved using the respective heuristics (P, R, V, C, or M), first only counting satisfiable, then unsatisfiable, and finally all instances. Best results for a benchmark set are indicated in boldface.

Whereas on satisfiable instances our restart heuristics did not perform better then the plain Minisat 2.0 RGR heuristic (in general even worse, with the exception of heuristic V on the SAT Competition 2007 instances, industrial category), on unsatisfiable instances our heuristics were able to outperform the static Minisat scheme. Notable is a

[4] As mentioned in the introduction, MiniSat 2.0 implements a RGR strategy, where times between restarts grows exponentially over time.

[5] Available from `http://www.miroslav-velev.com/sat_benchmarks.html`

[6] The SAT-Race 2006 and 2008 instances have been processed with the SatELite preprocessor.

100% increase in the number of solved instances for our heuristic P on the *velev-pipe* benchmark suite, as well as an increase of almost 10% on the *manolios* benchmarks. Considering both satisfiable and unsatisfiable instances, heuristic P still outperforms MiniSat in the number of solved instances in general.

On selected instances from the *manolios* and *velev-pipe* benchmark families, our restart scheme P considerably outperforms MiniSat, with speed-ups up to a factor of approximately 167. On average, heuristic P shows a speed-up of more than 38% over MiniSat 2.0 on these benchmark families. Noteworthy is also the achieved reduction in search space, measured in number of conflicts, which lies between 42.6% and 99.2%. In only one case (out of all 42 instances of the *manolios* benchmark that could be solved by both heuristics in between 2 and 15 minutes) the number of conflicts was higher for our strategy P. A reduction in number of conflicts is also typically connected with shorter proofs of unsatisfiability. This is especially important for algorithms that further process proofs generated by SAT solvers, like interpolation-based methods [13]. It is also striking that the number of restarts made by heuristic P is much higher than the number of restarts made by MiniSat.

We also made experiments comparing heuristic P with the latest, non-public version of MiniSat (2.1), as used in SAT-Race 2008. MiniSat 2.1 adds a whole set of improvements to version 2.0, including phase memorization, special handling for binary and blocked clauses, an improved memory manager, and a Luby restart strategy. MiniSat 2.1 has shown to perform considerably better than MiniSat 2.0 in SAT-Race 2008 (81 vs. 59 solved instances). This even holds when MiniSat 2.0 is extended with our new problem-sensitive restart strategy P. However, on the *manolios* benchmark family, our heuristic performs better, even without the other improvements implemented in MiniSat 2.1. Considering only unsatisfiable instances, strategy P is also better on the SAT-Race 2006 instances.

We performed further experiments with a modified version of MiniSat 2.0 implementing the same Luby strategy as in MiniSat 2.1. However, on all benchmark sets, we obtained better results with our strategy P. In another set of experiments we added phase memorization to MiniSat 2.0 with Luby strategy and heuristics P. Again, the results with strategy P were better (in number of solved instances), with the only exception of the *velev* benchmark set, where the Luby strategy was able to solve one instance more.

RViewer: A Tool for Monitoring Search Parameters. To experiment with different restart heuristics and to visualize search parameters over time we have implemented a Java tool called *RViewer*[7]. RViewer reads a dump file generated by a CDCL SAT solver, which contains the sequence of search parameters over time, one for each conflict. RViewer visualizes the development of the contained parameters during search. It allows to select one or multiple search parameters for display, computation of the moving average, zooming in and out, as well as moving through the dump file. RViewer was of great help in setting up restart policy P and to detect the phenomenon of plateaus for the backtrack level.

[7] RViewer is available for download at `http://baldur.iti.uka.de/software/RViewer`

4 Related Work and Conclusion

Huang [14] experimentally compares different restart policies, including a whole set of different RGR strategies, and a strategy based on Luby sequences. All policies examined in his survey are static ones, but he suggests that "substantial performance gains may be possible by using appropriate dynamic restart policies." Kautz *et al.* [15] give theoretical and empirical results on context-sensitive restart policies for randomized search procedures. The notion of context-sensitivity they use differs from our notion of problem-sensitivity in that their strategies are selected per instance, whereas our strategies are "more dynamic" in that they can also vary during a solver's run. Related to our work is that of Haim and Walsh [16], where they estimate the runtime of a SAT solver based on problem parameters observed during the initial phase of the search.

We have presented different dynamic, problem-sensitive restart heuristics that we implemented on top of MiniSat 2.0. We obtained good results with our policy P, which is based on avoidance of plateaus. The phenomenon of plateaus, which we observed in almost all SAT instances using our tool RViewer, also seems to be new. Directions for future research include further refinement of dynamic restart policies, especially by employing a combination of several problem parameters. It would also be interesting to find a theoretical underpinning why unsatisfiable instances seem to profit most from dynamic restart policies. A larger set of different restart policies might also be of help in implementing parallel SAT solvers based on competition parallelism.

References

1. Crawford, J.M., Baker, A.B.: Experimental results on the application of satisfiability algorithms to scheduling problems. In: AAAI (1994)
2. Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: AAAI/IAAI (1997)
3. Marques Silva, J.P., Sakallah, K.A.: GRASP – a new search algorithm for satisfiability. In: ICCAD (1996)
4. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC (2001)
5. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
6. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: AAAI/IAAI (1998)
7. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
8. Walsh, T.: Search in a small world. In: IJCAI (1996)
9. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Inf. Process. Lett. 47(4) (1993)
10. Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: AAAI/IAAI (2002)
11. Wei, W., Li, C.M., Zhang, H.: A switching criterion for intensification and diversification in local search for SAT. J. Satisfiability, Boolean Modeling and Comput. 4, 219–237 (2008)
12. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 28–33. Springer, Heidelberg (2008)

13. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
14. Huang, J.: The effect of restarts on the efficiency of clause learning. In: IJCAI (2007)
15. Kautz, H.A., Horvitz, E., Ruan, Y., Gomes, C.P., Selman, B.: Dynamic restart policies. In: AAAI/IAAI (2002)
16. Haim, S., Walsh, T.: Online estimation of SAT solving runtime. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 133–138. Springer, Heidelberg (2008)

(1,2)-QSAT: A Good Candidate for Understanding Phase Transitions Mechanisms

Nadia Creignou[1], Hervé Daudé[2], Uwe Egly[3], and Raphaël Rossignol[4]

[1] Université d'Aix-Marseille II, Laboratoire d'Informatique Fondamentale, Luminy, F-13288 Marseille, France
[2] Université d'Aix-Marseille I, Laboratoire d'Analyse, Topologie et Probabilités, Chateau Gombert F-13453 Marseille, France
[3] Institut für Informationsysteme 184/3, Technische Universität Wien Favoritenstrasse 9-11, A-1040 Wien, Austria
[4] Université de Paris 11, Département de Mathématiques, Bâtiment 425, F-91405 Orsay Cedex, France

Abstract. We explore random Boolean quantified CNF formulas of the form $\forall X \exists Y \varphi(X, Y)$, where X has $m = \lfloor \alpha \log n \rfloor$ variables ($\alpha > 0$), Y has n variables and each clause in φ has one literal from X and two from Y. These (1,2)-QCNF-formulas, which can be seen as quantified extended 2-CNF formulas, were introduced in SAT'08. It was proved that the threshold phenomenon associated to the satisfiability of such random formulas, (1,2)-QSAT, is controlled by the ratio c between the number of clauses and the number n of existential variables. In this paper, we prove that the threshold is sharp. For any value of α, we give the exact location of the associated critical ratio, $a(\alpha)$. At this ratio, our study highlights the sudden emergence of unsatisfiable formulas with a very specific shape. From the experimental point of view (1,2)-QSAT is challenging. Indeed, while for small values of m the critical ratio can be observed experimentally, it is not anymore the case for bigger values of m. For small values of m we give precise numerical estimates of the probability of satisfiability for critical (1,2)-QCNF-formulas. These experiments give evidence that the asymptotical regime is difficult to reach and provide some indication on the behavior of random instances. Moreover, experiments show that the computational effort, which is increasing with m, is maximized within the phase transition.

1 Introduction

In the last decades, numerous experimental studies have provided strong evidence that the difficulty to solve large instances of k-SAT is tightly linked to a phase transition in the probability that a random instance is satisfiable. As the clauses-to-variables ratio increases, the vast majority of formulas abruptly stop being satisfiable at a critical threshold point. The instances that are hard to solve seem to be located around this critical point. Determining the nature of the phase transition, locating it, determining a precise scaling window and gaining a better understanding of the structure of the space of solutions turn out to be

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 363–376, 2009.

challenging tasks, which have aroused a lot of interest and fructuous collaborations among different disciplines, namely combinatorics, probability, computer science and statistical physics.

Most of the studies have focused on 3-SAT. However, neither the value, nor even the existence of a critical ratio has been established. In order to gain insight into this hard problem, several researchers turned to studying a tractable variant: 2-SAT. Chvàtal and Reed [CR92], Goerdt [Goe96] and Fernandez de la Vega [dlV92] independently proved that 2-SAT exhibits a sharp transition. They proved that the critical clauses-to-variables ratio is equal to 1. More recently Bollobàs *et al.* [BBC+01] determined the scaling window for random 2-SAT. The reason why so many results could be obtained is the existence of a simple combinatorial characterization of unsatisfiable 2-CNF formulas [APT79]. On the one hand, this characterization provides an efficient (linear time) procedure to decide the satisfiability of 2-CNF formulas, thus allowing simulations at a very high scale. On the other hand, this characterization enables a direct combinatorial attack of random 2-SAT that focuses on the emergence of the most likely unsatisfiable formulas in the evolution of the random formula (see [CR92, Goe96]). In comparison, the difficulty to understand the phase transition for 3-SAT can be explained by the fact that 3-SAT is NP-complete (thus making simulations hard to run), and lacks a simple characterization of unsatisfiable formulas.

We propose to investigate an intermediate satisfiability problem, (1,2)-QSAT, which was first introduced by the authors in [CDER08]. This problem concerns a certain subclass of quantified Boolean formulas. More precisely, we are interested in (1,2)-QCNF-formulas, namely in formulas of the type $\forall X \exists Y \varphi(X, Y)$, where X has m variables, Y has n variables, and $\varphi(X, Y)$ is a conjunction of 3-clauses, each of which containing exactly one universal literal and two existential ones. This problem has several interesting features, which make it a good candidate to understand the mechanisms driving phase transitions. The key is that a (1,2)-QCNF-formula can be seen as an extended 2-CNF-formula. Indeed, the semantical elimination of the $\forall$-quantifiers yields the conjunction of 2^m existential 2-CNF formulas. As a first consequence this gives an upper bound for the complexity of deciding the truth value of such a formula, $O(2^m|\varphi|)$. This means that this problem can be solved in polynomial time when m is of logarithmic order compared to n (nevertheless, it is worth noticing that, in full generality, this problem is coNP-complete, see [FKB90]). Thus, the additional parameter of this problem, m, makes the complexity of the problem scalable to a polynomial of any degree. Moreover, since it is a quantified satisfiability problem we can take advantage of competitive QBF solvers for running experiments. As a second consequence, it inherits the good combinatorial properties of 2-SAT: there is a simple combinatorial characterization of unsatisfiable (1,2)-QCNF-formulas. So, we have here a problem much harder to solve than 2-SAT, and for which we can nevertheless hope to combine successfully practical and theoretical studies on random instances.

In [CDER08], we proved that the phase transition for (1,2)-QSAT is controlled by the clauses-to-existential-variables ratio c. We showed that when m is small

enough, there is a critical ratio equal to 2, when m is big enough there is a critical ratio equal to 1. When m is of logarithmic order compared to n, $m = \lfloor \alpha \ln n \rfloor$ ($\alpha > 0$), there is an intermediate regime. At this intermediate regime, we obtained lower and upper bounds for the critical ratio, whenever it exists, thus showing that it is strictly between 1 and 2. It was left open the nature of the transition and the exact value of the critical ratio, whenever it exists. It was also observed that, while the simulations had turned out to be very helpful to guide the theoretical investigation, they failed in giving a precise location of the threshold.

In this paper, we get a new upper bound for the threshold, which matches the lower bound given in [CDER08]. Thus, we prove that the threshold is sharp. For any value of α, we give the exact location of the associated critical ratio, $a(\alpha)$. Moreover, we prove the emergence of very typical minimal unsatisfiable subformulas when entering the unsatisfiability phase.

From the experimental side, we continue our simulations. We run experiments with a much higher precision than before. While in [CDER08] we used 1000 formulas per data point, we use here 100,000 formulas. As a result, the numerical estimates that are obtained have a much lower uncertainty. We report extended simulations for small values of m. They allow to give reliable numerical estimates of the probability of satisfiability of critical (1,2)-QCNF-formulas (i.e., formulas having exactly $a(\alpha) \cdot n$ clauses). They indicate that this probability decreases to 0 as m increases. Also, they give strong evidence that, for $m \geq 5$, the experiments are not run at a scale high enough in order to deliver results which also hold for very large problem instances. Finally they provide evidence that at least when $m \leq 4$ and n is big enough, a random (1,2)-QCNF-formula behaves as the conjunction of 2^m independent random 2-CNF-formulas. Moreover, we measure the computational effort which is needed to decide the truth value of random instances. We show that these instances exhibit a now well-known "easy-hard-easy" pattern.

2 Theoretical Results

2.1 Definition of the Problem and Main Result

A *literal* is a propositional variable or its negation. The *atom* of a literal l is the variable p if l is p or $\overline{p}$. Literals are said to be *strictly distinct* when their corresponding atoms are pairwise different. A *clause* is a finite disjunction of literals. A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. A formula is in k-CNF, if any clause consists of exactly k literals. Here we are interested in formulas of the form

$$F = \forall X \exists Y \varphi(X, Y)$$

where $X = \{x_1, \ldots, x_m\}$, and $Y = \{y_1, \ldots, y_n\}$, and $\varphi(X, Y)$ is a 3-CNF formula with exactly one universal and two existential literals in each clause. We will call such formulas (1,2)-QCNFs.

A truth assignment for the existential (resp. universal) variables, Y (resp. X) is a Boolean function $I : Y \to \{0,1\}$ (resp. $X \to \{0,1\}$), which can be extended to literals by $I(\overline{x}) = 1 - I(x)$. A (1,2)-QCNF formula is *true* (or *satisfiable*) if for every assignment to the variables X, there exists an assignment to the variables Y such that φ is true.

We consider formulas built on m universal variables and n existential variables. Thus we have

$$N = m\binom{n}{2}2^3 = 4mn(n-1)$$

different clauses at hand. We may establish our result in considering random formulas obtained by taking each one of the N possible clauses independently from the others with probability $p \in]0,1[$. Let $c > 0$, it is well known (see for instance [JLR00, Sections 1.4 and 1.5]) that the threshold obtained in this model translates to the model alluded to in the introduction – in which $L = \lfloor cn \rfloor$ distinct clauses are picked uniformly at random among all the N possible choices – when $p = \frac{L}{4mn(n-1)}$. Thus, from now on we shall always suppose that $p = \frac{c}{4mn}$, and we continue to denote by $\mathbb{P}_{m,c}(n)$ the probability that a random formula in this model is satisfiable. We are interested in studying $\lim_{n\to+\infty} \mathbb{P}_{m,c}(n)$ as a function of the parameters m and c. Any value of c such that $\mathbb{P}_{m,c}(n) \to 1$ (resp. such that $\mathbb{P}_{m,c}(n) \to 0$) gives a lower (resp. upper) bound for the threshold effect associated to the phase transition.

In [CDER08], it has been established that when m is small enough, actually when $m \leq \dfrac{\log n}{\log 2}$, there is a sharp threshold at $c = 2$. On the other side, when m is large enough, actually when $m \gg \ln n$, there is a sharp threshold at $c = 1$. Moreover, when $m = \lfloor \alpha \ln n \rfloor$ with $\alpha > \dfrac{1}{\ln 2}$, an upper bound $a(\alpha)$ and a lower bound $b(\alpha)$ for the location of the transition have been given: $a(\alpha)$ being the solution of the equation $\alpha \cdot H(c) = 1$, where $H(c) = \ln(c) + \left(\dfrac{2}{c} - 1\right)\ln(2-c)$ and $b(\alpha)$ being strictly greater than $a(\alpha)$. Thus the following proposition was established.

Proposition 1. [CDER08]
For any $\dfrac{1}{\ln 2} < \alpha$, *if* $c < a(\alpha)$, *then* $\mathbb{P}_{\lfloor \alpha \ln n \rfloor, c} \xrightarrow[n\to+\infty]{} 1$

Here, in developing further the techniques used by Chvàtal and Reed [CR92] and Goerdt [Goe96], we prove the following.

Theorem 1. *For any* $\dfrac{1}{\ln 2} < \alpha$, *if* $c > a(\alpha)$, *then* $\mathbb{P}_{\lfloor \alpha \ln n \rfloor, c} \xrightarrow[n\to+\infty]{} 0$.

Thus, we can plot in Fig. 1 the evolution of the critical ratio as a function of α.

Our analysis is based on new results about specific minimal unsatisfiable formulas, called pure snakes.

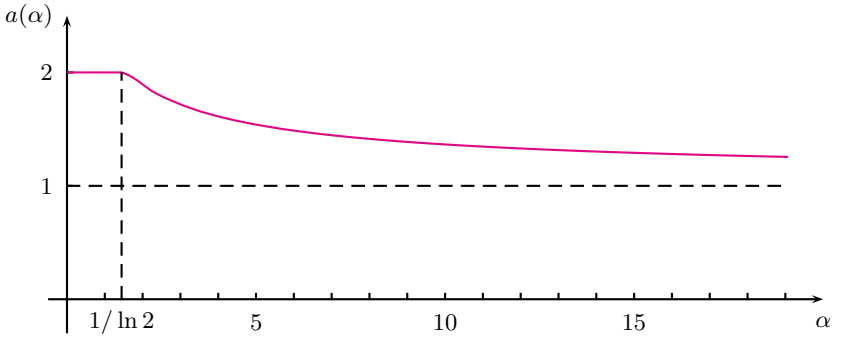


Fig. 1. Evolution of the critical ratio values

2.2 Pure Snakes

The notion of purity over sets of universal literals is useful to characterize the truth value of (1,2)-QCNF-formulas. A (multi-)set or sequence of literals is *pure* if it does not contain both a variable x and its negation $\overline{x}$.

Definition 1. *A pure* snake *of length $s+1 \geq 4$, with $s+1 = 2t$, is a set of $s+1$ clauses $C_0, \ldots, C_s$ which have the following structure: there are s strictly distinct existential literals $w_1, \ldots, w_s$, and a pure sequence of $s+1$ universal literals $v_0, \ldots, v_s$ such that, for every $0 \leq r \leq s$, $C_r = (v_r \vee \overline{w_r} \vee w_{r+1})$ with $w_0 = w_{s+1} = \overline{w_t}$.*

Every (1,2)-QCNF-formula that contains a pure snake is false. Let $X_{s,k}$ be the number of pure snakes of length $s+1$ with k universal variables in a random (1,2)-QCNF formula. We obtain

$$\mathbb{E}_{m,c}(X_{s,k}) = p^{s+1} \cdot (n)_s \cdot 2^s \cdot \binom{m}{k} \cdot 2^k \cdot \mathcal{S}(s+1,k) \cdot k! \tag{1}$$

with $\mathcal{S}(s+1,k)$ denoting the Stirling number of the second kind and $(n)_s = n(n-1)\cdots(n-s+1)$.

When $m = \lfloor \alpha \ln n \rfloor$, it appears that long snakes of length $\gg \ln n$ have asymptotically no chance to appear when $\alpha > 1/\ln 2$ and $c \in]1,2[$. Therefore, in our study we can focus on snakes of length proportional to $\ln n$. Hence, let us set $\beta = k/\ln n$ and $\gamma = (s+1)/\ln n$.

The following proposition shows that the behavior of the average number of snakes is governed by a continuous function of several real variables. It can be derived in using the following well-known estimates for binomial coefficients and precise results on Stirling numbers of the second kind. If $1 \leq b \leq a$, then

$$\sqrt{\frac{1}{a}} \left(\frac{a}{b}\right)^b \cdot \left(\frac{a}{a-b}\right)^{a-b} \leq \binom{a}{b} \leq \left(\frac{a}{b}\right)^b \cdot \left(\frac{a}{a-b}\right)^{a-b} . \tag{2}$$

Moreover, as shown in [Tem93], there exist $K > 0$ and $K' > 0$ such that, for $1 \leq b \leq a$,

$$K\sqrt{\frac{b}{a}}\left(\frac{e^{x_0}-1}{x_0}\right)^b\left(\frac{a}{e}\right)^a x_0^{b-a} \leq b!\mathcal{S}(a,b) \leq K'\sqrt{b}\left(\frac{e^{x_0}-1}{x_0}\right)^b\left(\frac{a}{e}\right)^a x_0^{b-a} \quad (3)$$

where $x_0 > 0$ is a function of b/a defined implicitly for $b < a$ by $1 - e^{-x_0} = \frac{b}{a}x_0$, and for $a = b$ by $x_0 = 0$. The conventions are that $0^0 = 1$ and $\frac{e^0-1}{0} = 1$.

Proposition 2. *There exist $A > 0$ and $B > 0$ such that for any $c > 0$, for every positive integers n, m, s and k such that $k \leq \min(m, s+1)$, it holds that*

$$\frac{A\,(n)_s\sqrt{k}}{n^s\sqrt{m(s+1)}}\; n^{g_{\frac{m}{\ln n},c}(\frac{k}{\ln n},\frac{s+1}{\ln n})} \leq \mathbb{E}_{m,c}(X_{s,k}) \leq B\sqrt{m}\; n^{g_{\frac{m}{\ln n},c}(\frac{k}{\ln n},\frac{s+1}{\ln n})} \quad (4)$$

where for any $1 < c < 2$ and $\alpha > 0$, $g_{\alpha,c}$ is a continuous function on $\mathcal{D}_\alpha = \{(\beta,\gamma) \mid 0 < \beta \leq \alpha \text{ and } \beta \leq \gamma\}$ with a strict and global maximum on $\mathcal{D}_\alpha$, given by its unique stationary point in $\mathcal{D}_\alpha$. More precisely

$$\max_{\mathcal{D}_\alpha} g_{\alpha,c}(\beta,\gamma) = g_{\alpha,c}(\hat{\beta}(\alpha,c),\hat{\gamma}(\alpha,c)) = \alpha H(c) - 1 \quad (5)$$

with $\hat{\beta} = \dfrac{2\alpha(c-1)}{c}$, $\hat{\gamma} = \dfrac{-2\alpha\ln(2-c)}{c}$, $H(c) = \ln c + \left(\dfrac{2}{c} - 1\right)\ln(2-c)$.

This result points out for each α the values of k and s that contribute the most to the average number.

2.3 Dominant Pure Snakes at the Phase Transition

The proof of our main result (Theorem 1) is obtained in considering pure snakes of a very specific shape together with a general exponential inequality. More precisely, let $s+1 = \lfloor\hat{\gamma}\ln n\rfloor = 2t$ and $k = \lfloor\hat{\beta}\ln n\rfloor$. From [JLR00, Theorem 2.18 ii)], we know that

$$\mathbb{P}_{m,c}(n) \leq \Pr(X_{s,k} = 0) \leq \exp\left(-\frac{\mathbb{E}_{m,c}(X_{s,k})}{1 + \sum_{i=1}^{s} N_{m,s,k}(i)p^{s+1-i}}\right) \quad (6)$$

where $N_{m,s,k}(i)$ denotes the number of pure snakes B of length $s+1$ with k universal variables that share exactly i clauses with a given pure snake A_0 of length $s+1$ with k universal variables. Starting from Proposition 2, we can derive some tight bounds for the quantities appearing in the right-hand side of (6). Thus, it can be shown that, when $\alpha\ln 2 > 1$ and for any $c > a(\alpha)$, we have $\mathbb{P}_{m,c}(n) = o(1)$. More precisely we also gain new and interesting information about the structure of random unsatisfiable formulas at the threshold ratio, as stated in the following proposition.

Proposition 3. *Let us denote by $A_{s,k}$ the event that there exists a pure snake of length $s+1$ with k universal variables. For any $\alpha > 1/\ln 2$, define:*

$$\beta(\alpha) = \frac{2\alpha(a(\alpha)-1)}{a(\alpha)},$$

$$\gamma(\alpha) = -2\alpha\ln(2-a(\alpha))/a(\alpha) \quad \textit{and}$$

$$I(\alpha) = [(\gamma(\alpha)-\varepsilon_n)\ln n, (\gamma(\alpha)+\varepsilon_n)\ln n] \times [(\beta(\alpha)-\varepsilon_n)\ln n, (\beta(\alpha)+\varepsilon_n)\ln n] \ .$$

Then, for any $\alpha > 1/\ln 2$, there are suitable positive constants C_1 and C_2, such that for $\delta_n = C_1\frac{\ln\ln n}{\ln n}$, $\varepsilon_n = C_2\sqrt{\delta_n}$, $m = \lfloor \alpha \ln n \rfloor$ and $c_n = a(\alpha)+\delta_n$:

$$\mathbb{P}_{m,c_n}\left(\bigcup_{(s,k)\in I(\alpha)^c} A_{s,k}\right) = o(1)$$

and

$$\mathbb{P}_{m,c_n}\left(A_{\lfloor\gamma(\alpha)\ln n\rfloor,\lfloor\beta(\alpha)\ln n\rfloor}\right) = 1-o(1) \ .$$

Thus we prove that when we enter the phase of unsatisfiability, pure snakes suddenly appear with a very specific structure, namely their number of universal variables is $\beta(\alpha)\ln n$, and their number of existential variables (and hence, their length) is $\gamma(\alpha)\ln n$, where $\beta(\alpha)$ and $\gamma(\alpha)$ are positive functions of α.

3 Experimental Results

3.1 The Threshold for Small Values of m

Before we start discussing the empirical results, let us first describe how we ran the experiments. All experiments were conducted according to the same scheme, which is described with the help of Fig. 2. One experiment consisted in generating at random (in drawing uniformly and independently) (1,2)-QCNF formulas over given values of m universal variables and n existential variables, with a clauses-to-existential-variables ratio indicated by the points. Since we are interested in the behavior around the crossing point of the curves, we draw them up to a ratio between 2.1 and 2.2, regardless whether the curves have reached the x-axis or not. In Fig. 2, we considered successively $m = 1, 2, 3$ and 4 and $n = 2,000$, $4,000$, $8,000$, $16,000$ and $32,000$. For each of the chosen values of ratio, a sample of $100,000$ formulas was studied using the QBF solver QuBE [GNT01], thus computing the truth value of each formula. The proportion of true (or satisfiable) instances for each considered value of ratio was then plotted.

Let us recall that for constant values of m, the critical ratio is equal to 2. This can be observed in Fig. 2. Indeed, in each of these figures, the curves sharpen as n increases and pivot about a single point, thus indicating a critical ratio at $c = 2$. Already when $m = 4$, the crossing point seems to occur before 2.

In Fig. 3, one can observe that the crossing point is moving to the left as m increases. For $m = 7$, one can see successive crossings of pairs of curves

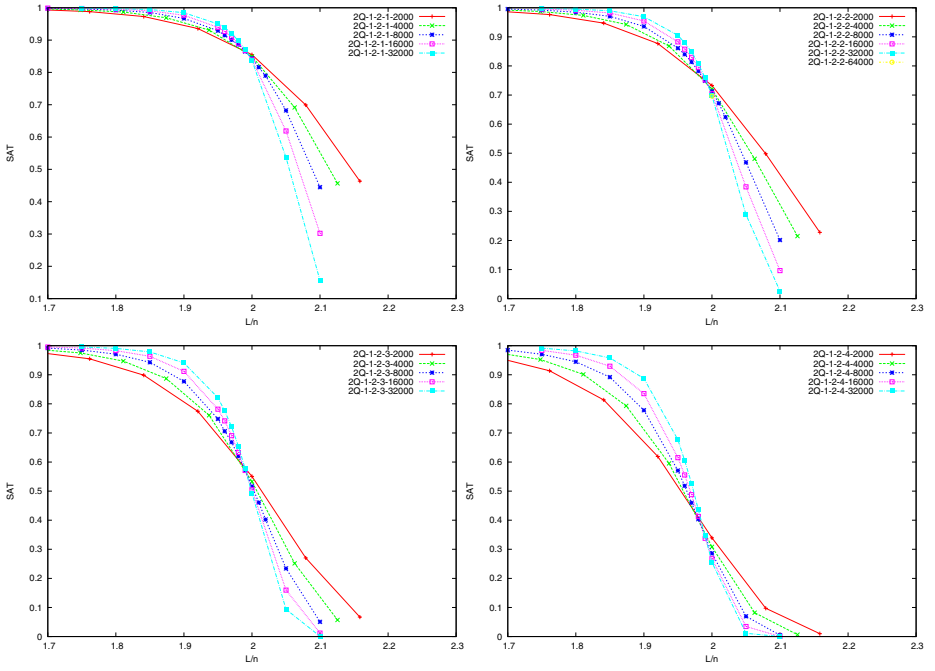


Fig. 2. The satisfiability curves for $m = 1$, $m = 2$, $m = 3$ and $m = 4$

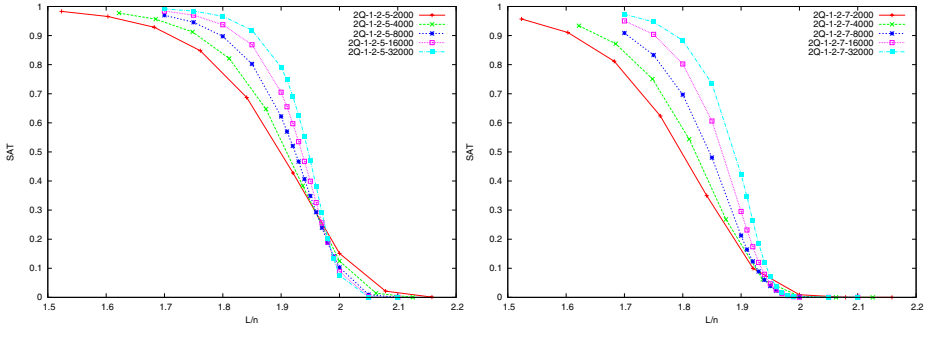


Fig. 3. The satisfiability curves for $m = 5$ and $m = 7$

for increasing values of n, thus making a rough estimate of the critical ratio difficult. Observe moreover that the probability of satisfiability exactly at the critical ratio, i.e., when the number of clauses is $2n$, seems to vanish to 0 as m increases. So, these observations suggest that the critical value is difficult to estimate from the experiments as m increases for two reasons: the probability of satisfiability for critical formulas vanishes to 0, while the asymptotical regime is reached at a higher scale, as m increases.

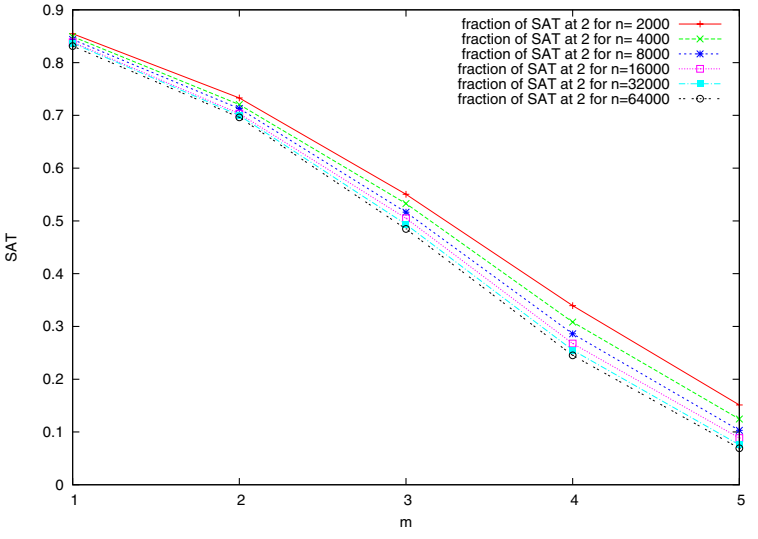


Fig. 4. The fraction of satisfiable formulas at $c = 2$ for different n and m

This will be confirmed by numerical estimates of the probability of satisfiability for critical formulas as discussed in the next section.

3.2 Numerical Estimates of the Probability of Satisfiability for Critical (1,2)-QCNF Formulas

Numerical estimates of the probability of satisfiability of a critical random 2-CNF formula, i.e., a random 2-CNF formula whose number of clauses is equal to the number of variables, were given in [DM06]. We give here similar estimates for (1,2)-QSAT for small values of m. For $m = 1$ up to 5, we provide precise results based on numerical estimates of the probability of satisfiability for (1,2)-QCNF-formulas with $n = 2,000$ up to $32,000$ existential variables, with a critical ratio equal to 2. Recall that, for each size n, we drew at random 100,000 instances and we determined if they were satisfiable or not. Results from the simulations are plotted in Fig. 4. It is worth noticing that the high number of instances we used for each data point is needed in order to get reliable statistics as discussed below.

These estimates indicate that for m fixed and small, the probability of satisfiability of a random critical (1,2)-QCNF-formula is converging to a positive value as n tends to infinity and that this value decreases as m increases. Moreover, observe that, for a fixed m, this probability decreases as n increases. For m =1, this probability of satisfiability seems to have reached a fixed point (at $m = 1$ the five data points are very close to each other). On the contrary, for $m = 5$ the probability does still significantly decrease when n is increased and has not reached a fixed point. This gives evidence that, for $m = 5$, we have not yet entered the asymptotical regime.

Table 1.

m	n	$\hat{p}_{0,m}(n)$	$\hat{p}_1(n,m)$	$\hat{I}_{m,n}$	fit
1	8000	0.84723	0.84379	] -0.01203,0.00515 [	y
1	16000	0.84217	0.83879	] -0.01197,0.00521 [	y
1	32000	0.83856	0.83683	] -0.01032,0.00686 [	y
1	64000	0.83678	0.83133	] -0.01404,0.00314 [	y
2	8000	0.71952	0.71298	] -0.01513,0.00205 [	y
2	16000	0.71008	0.70332	] -0.01535,0.00183 [	y
2	32000	0.70324	0.69994	] -0.01189,0.00529 [	y
2	64000	0.6969	0.6962	] -0.00929,0.00789 [	y
3	8000	0.51781	0.51649	] -0.00991,0.00727 [	y
3	16000	0.50668	0.50504	] -0.01023,0.00695 [	y
3	32000	0.49447	0.49337	] -0.00969,0.00749 [	y
3	64000	0.48683	0.4847	] -0.01072,0.00646 [	y
4	8000	0.2673	0.2861	] 0.01021,0.02739 [	n
4	16000	0.25349	0.26775	] 0.00567,0.02285 [	n
4	32000	0.2461	0.2551	] 0.00041,0.01759 [	n
4	64000	0.23761	0.24531	] -0.00089,0.01629 [	y
5	8000	0.07213	0.10304	] 0.02232,0.03950 [	n
5	16000	0.06411	0.08851	] 0.01581,0.03299 [	n
5	32000	0.06039	0.07561	] 0.00663,0.02381 [	n
5	64000	0.05596	0.06914	] 0.00459,0.02177 [	n

Actually, these estimates give us another important indication on the behavior of random critical instances for small m. Let $p_0(n)$ be the probability that a random 2-CNF formula with n variables and n clauses is satisfiable. Let $p_1(n,m)$ be the probability that a random (1,2)-QCNF-formula with n existential variables, m universal variables and $2n$ clauses is satisfiable. For each $m > 0$, the semantical elimination of the $\forall$-quantifiers in such a formula yields the conjunction of 2^m 2-CNF formulas. More precisely, for $m = 1$, it can be shown that a random (1,2)-QCNF-formula with parameter c behaves as the conjunction of two *independent* random 2-CNF-formula with parameter equivalent (as n goes to infinity) to $c/2$. Thus, one has:

$$p_1(1,n) - p_0(n)^2 \xrightarrow[n\to\infty]{} 0 \ .$$

When $m \geq 2$, a random (1,2)-QCNF-formula with parameter c behaves as the conjunction of 2^m *dependent* random 2-CNF-formula with parameter equivalent to $c/2$. In fact, each pair of such "half-formulas" contains, roughly, between 0 and $\frac{m-1}{m}c/2$ clauses in common. However, it seems plausible that this dependence is not important concerning satisfiability at the threshold, i.e, at $c = 2$ for fixed m. Indeed, for critical 2-SAT, it is plausible that knowing that a formula F is true (resp. false) does not give a significant hint about the fact that a formula F' obtained from F by re-sampling a fixed fraction of its clauses is still true (resp. false). This would lead (somewhat rashly) to conjecture that $p_1(m,n) - p_0(n)^{2^m}$

goes to zero as n goes to infinity, for fixed m. To support this conjecture, we give estimations for $p_1(m,n) - p_0(n)^{2^m}$.

For some values of n and m, we drew at random $N = 100,000$ conjunctions of 2^m critical 2-CNF-formulas, i.e., with critical ratio $c_0 = 1$. We counted the proportion $\hat{p}_{0,m}(n)$ of satisfiable conjunctions. Next, for the same values of n and m, we drew at random $N = 100,000$ critical (1,2)-QCNF-formulas, i.e., with critical ratio $c_1 = 2$. We counted the proportion $\hat{p}_1(m,n)$ of satisfiable formulas. We want to know, on the basis of these estimators, whether it is plausible that $p_1(m,n) - p_0(n)^{2^m}$ goes to zero as n goes to infinity. Hoeffding's inequality (see [Hoe63]) implies that for any $\varepsilon \in]0,1[$, m and n,

$$\mathsf{Pr}\left(|\hat{p}_1(m,n) - \hat{p}_{0,m}(n) - (p_1(m,n) - p_0(n)^{2^m})| \geq \sqrt{\frac{2}{N} \log \frac{2}{\varepsilon}} \right) \leq \varepsilon \ ,$$

which gives us the following confidence interval for $p_1(m,n) - p_0(n)^{2^m}$ with security coefficient $1 - \varepsilon$:

$$\hat{I}_{m,n} = \left] \hat{p}_1(m,n) - \hat{p}_{0,m}(n) - \sqrt{\frac{2}{N} \log \frac{2}{\varepsilon}}; \hat{p}_1(m,n) - \hat{p}_{0,m}(n) + \sqrt{\frac{2}{N} \log \frac{2}{\varepsilon}} \right[\ .$$

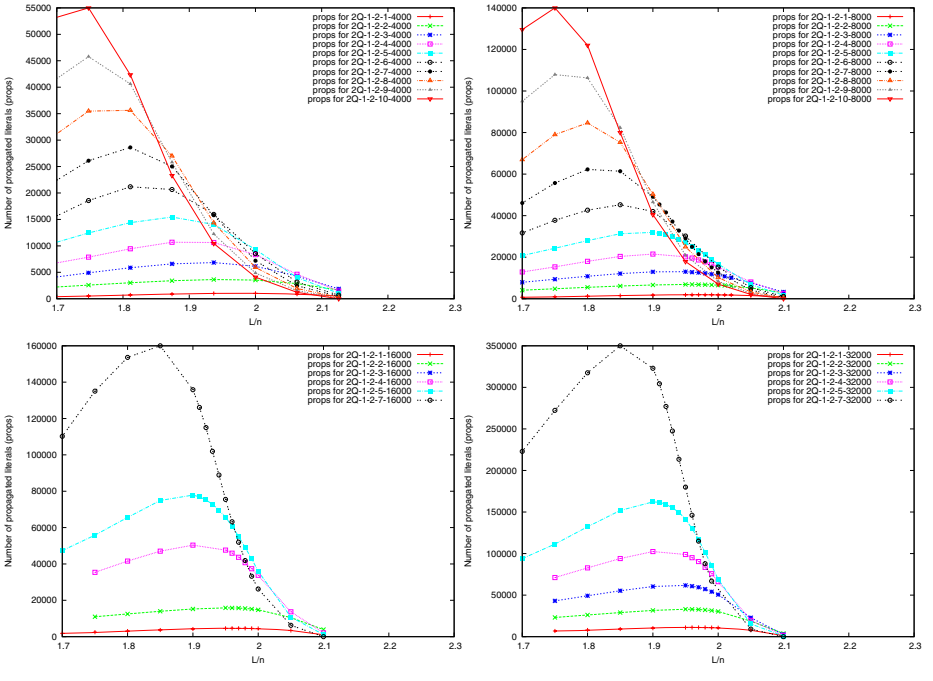


Fig. 5. The propagation curves for $n = 4,000; 8,000; 16,000$ and $32,000$

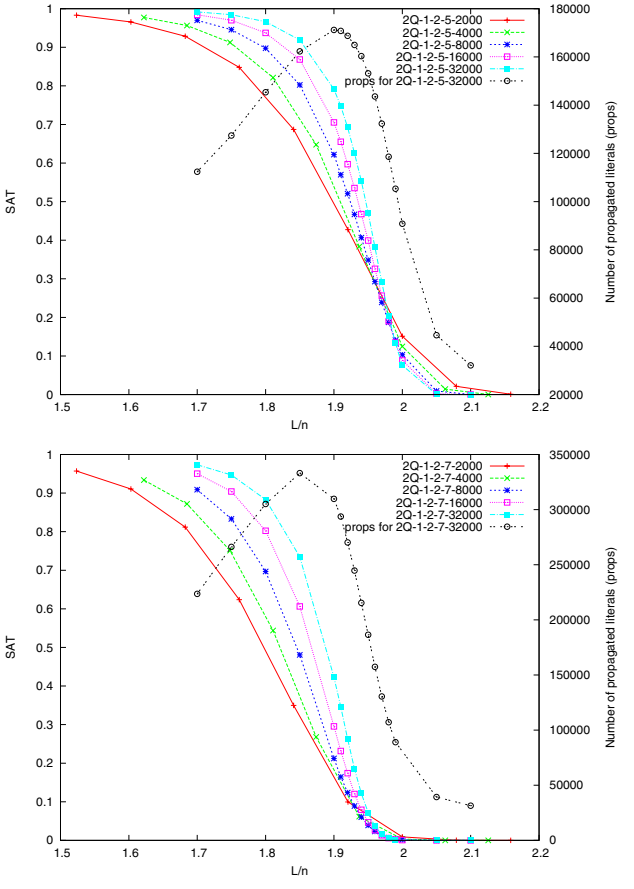


Fig. 6. A propagation curve together with satisfiability curves for $m = 5$ and $m = 7$

Choosing $\varepsilon = 0.05$, we consider that our conjecture is supported by the simulations if, for n large enough, the interval $\hat{I}_{m,n}$ contains zero. This is reported by a "y" in the column "fit" of Table 1.

We can observe that it fits for $m = 1, 2, 3$ and 4 for n large enough. The data starts to diverge for $m = 5$. There are two possible reasons: either the conjecture is false for $m = 5$, or $n = 64000$ is not large enough to see the convergence conjectured for this value of m. Notice however that it is close to fit for large n. In any case, these results support our conjecture that for m small enough (i.e, at least when $m \leq 4$), a critical (1,2)-QCNF-formula behaves as the conjunction of 2^m independent 2-CNF-formulas.

3.3 Where the Hard Instances Are

In [SML96], Selman *et al.* gave experimental evidence suggesting that there is a range of the clauses-to-variables ratio, r, within which it seems hard to decide

whether a randomly chosen 3-SAT instance is satisfiable or not. For $r \sim 4.2$, a satisfying assignment can be found for roughly half the formulas and around this point the computational effort for finding a truth assignment, whenever one exists, is maximized. We are interested in knowing whether such a pattern can also be observed for (1,2)-QSAT.

Measuring the running time of a program is a difficult task in a multi-core/multi-processor environment, especially if the machine has heavy load. Therefore, we have to use another measure like the number of branches, the number of decisions, etc. A measure which is offered by the used solver QuBE is the number of of propagations of (existential and universal) literals.

In Fig. 5, we find the curves showing the average number of propagations for a specific $n = 4000, 8000, 16000, 32000$ and different m. Notice that the scale on the y-axis is not the same from one figure to the other.

These results appear to be fully consistent with the worst-case complexity upper bound we have, i.e., the complexity naturally increases with m. Moreover, there is a peak in the computational effort. Interestingly, Fig. 6 shows that this peak occurs within the phase transition, i.e., the computational effort for deciding the truth value of a formula is maximized for when the percentage of true formulas is between 30 percent and 80 percent.

4 Conclusion

Our study of (1,2)-QSAT turns out to be even more challenging than expected. At the interesting regime, i.e., when $m = \lfloor \alpha \ln n \rfloor$ with $\alpha > 1/\ln 2$, we are able to give the exact location of the threshold as a function of α. Actually, our study goes beyond this step since we also give precise information on typical substructures that occur in random formulas just after the phase transition. Indeed, our analysis shows the emergence of specific minimally unsatisfiable formulas in evolving random formulas. Hence, we assert that, at the critical point c, pure snakes of length $-2\alpha \ln(2-c)/c \log n$ are the first syndrome of unsatisfiability.

We have made precise simulations for small m. They have given some indications on the probability of satisfiability of critical formulas. Obviously getting precise results as observing the critical ratio, or measuring the width of the transition from satisfiability to unsatisfiability (see [Wil02]) will require experiments at a much higher scale, and thus will need more computational power. At the same time, it is also challenging to find out new forms of simulations, whose results can guide the theoretical investigations, even if not run at a very high scale.

Finally, let us emphasize that (1,2)-QSAT is a quantified problem. Therefore, we can hope that its investigation will enable a better understanding of the typical behavior of random quantified formulas, and thus support the development of competitive QBF solvers. Let us recall that (1,2)-QSAT is a problem whose complexity smoothly interpolates between P and coNP. Moreover experiments have shown that random instances of (1,2)-QSAT exhibit an "easy-hard-easy"

pattern. The peak of the computational effort occurs within the phase transition Therefore, in making m varying, we get a whole bunch of instances having different complexities that we can be used to test and evaluate QBF solvers.

References

[APT79] Aspvall, B., Plass, M.F., Tarjan, R.E.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Information Processing Letters 8(3), 121–123 (1979)

[BBC+01] Bollobás, B., Borgs, C., Chayes, J.T., Kim, J.H., Wilson, D.B.: The scaling window of the 2-SAT transition. Random Structures and Algorithms 18(3), 201–256 (2001)

[CDER08] Creignou, N., Daudé, H., Egly, U., Rossignol, R.: New results on the phase transition for random quantified Boolean formulas. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 34–47. Springer, Heidelberg (2008)

[CR92] Chvátal, V., Reed, B.: Mick gets some (the odds are on his side). In: Proceedings of the 33rd Annual Symposium on Foundations of Computer Science (FOCS 1992), pp. 620–627 (1992)

[dlV92] de la Fernandez Vega, W.: On random 2-SAT (manuscript, 1992)

[DM06] Deroulers, C., Monasson, R.: Criticality and universality in the unit-propagation search rule. Eur. Phys. J. B 49, 339–369 (2006)

[FKB90] Flögel, A., Karpinski, M., Kleine Büning, H.: Subclasses of quantified Boolean formulas. In: Proceedings of the 4th Workshop on Computer Science Logic (CSL 1990), pp. 145–155 (1990)

[GNT01] Giunchiglia, E., Narizzano, M., Tacchella, A.: QuBE: A system for deciding quantified Boolean formulas satisfiability. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 364–369. Springer, Heidelberg (2001)

[Goe96] Goerdt, A.: A threshold for unsatisfiability. Journal of of Computer and System Sciences 53(3), 469–486 (1996)

[Hoe63] Hoeffding, W.: Probability inequalities for sums of bounded random variables. J. Amer. Statist. Assoc. 58, 13–30 (1963)

[JLR00] Janson, S., Luczack, T., Rucinski, A.: Random graphs. John Wiley, New York (2000)

[SML96] Selman, B., Mitchell, D., Levesque, H.J.: Generating hard satisfiability problems. Artificial Intelligence 81(1-2), 17–29 (1996)

[Tem93] Temme, N.M.: Asymptotic estimates of Stirling numbers. Stud. appl. Math. 89, 223–243 (1993)

[Wil02] Wilson, D.B.: On the critical exponents of random k-SAT. Random Structures and Algorithms 21(2), 182–195 (2002)

VARSAT: Integrating Novel Probabilistic Inference Techniques with DPLL Search

Eric I. Hsu and Sheila A. McIlraith

Department of Computer Science
University of Toronto
{eihsu,sheila}@cs.toronto.edu
http://www.cs.toronto.edu/~eihsu/VARSAT

Abstract. Probabilistic inference techniques can be used to estimate variable *bias*, or the proportion of solutions to a given SAT problem that fix a variable positively or negatively. Methods like Belief Propagation (BP), Survey Propagation (SP), and Expectation Maximization BP (EMBP) have been used to guess solutions directly, but intuitively they should also prove useful as variable- and value- ordering heuristics within full backtracking (DPLL) search. Here we report on practical design issues for realizing this intuition in the VARSAT system, which is built upon the full-featured MiniSat solver. A second, algorithmic, contribution is to present four novel inference techniques that combine BP/SP models with local/global consistency constraints via the EMBP framework. Empirically, we can also report exponential speed-up over existing complete methods, for random problems at the critically-constrained phase transition region in problem hardness. For industrial problems, VARSAT is slower that MiniSat, but comparable in the number and types problems it is able to solve.

Keywords: Probabilistic Inference, Survey Propagation/EMBP, Variable/Value Ordering Heuristics.

1 Introduction

A variety of message-passing (*a.k.a.* "propagation") algorithms have been used to estimate variable "bias," or the probability of finding each variable set one way or another if we could somehow sample from the space of solutions to a given SAT instance. In particular, Belief Propagation (BP) has been used to differentiate solutions where a variable is constrained to be positive from those where it is negatively constrained [1]. Survey Propagation (SP) extends this model to represent a third probability representing solutions where the variable is not constrained at all; on hypothetically sampling we might find that it is set positively or negatively, but flipping it would still result in a solution [2]. Either model can be employed within the Expectation Maximization Belief Propagation (EMBP) framework, a convergent alternative that accommodates a choice of consistency constraints for balancing speed and accuracy in estimating bias [3].

While bias estimators produce intuitively useful information about the solutions of SAT instances, they cannot actually solve a problem on their own. To date they have all employed a "decimation" framework of computing a single sequence of estimates,

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 377–390, 2009.

while setting a block of one or more most strongly-biased variables after each estimate. Taken with this decimation framework, the probabilistic bias estimators are state-of-the-art for solving large random problems near the critically-constrained phase transition in problem hardness [4]. However, this construction cannot backtrack or take advantage of modern advances in systematic DPLL search like clause learning; the decimation process either directly reaches a solution by a series of fortuitous variable assignments, or it ends in failure without determining satisfiability or unsatisfiability. Toward the upper reaches of the phase transition threshold, failure occurs about half the time (at $\alpha = 4.4$ for satisfiable problems). For industrial SAT problems with "real-world" structure, the combination is entirely unusable [5].

Here we report on the integration of six bias estimators as variable- and value-ordering heuristics within the full-featured MiniSat backtracking solver [6]. Moving past basic intuitions and creating a practical solver requires a number of possibly intuitive, but still non-obvious design decisions and optimizations, due especially to the high computational expense associated with bias estimation. A second contribution consists of four novel estimation techniques whose stand-alone accuracy has been summarized elsewhere without explanation–here they are presented in full for the first time [5]. In particular, we employ local or global consistency approximations to extend either the BP or the SP model, producing EMBP-L, EMSP-L, EMBP-G, and EMSP-G. Together with basic BP and SP, these rules drive the resulting VARSAT solver, so-named in recognition of the variational methods underlying the estimators [7]. VARSAT retains the superior performance of probabilistic techniques on hard random problems, but represents a first step toward handling industrial problems as well. On the latter, its performance is comparable to regular MiniSat in that they are both able to solve mostly the same problems within a given time limit; the main difference is that VARSAT is slower to solve the same problems.

The following two sections provide further background and formal definitions concerning propagation algorithms and their application to SAT. Next, Section 4 presents the six bias estimators alongside intuitive explanations of how they work. Then, Section 5 discusses their integration within backtracking DPLL search, and Section 6 summarizes the main experimental findings. Lastly, Section 7 extracts overall conclusions and discusses future work.

2 Background

Message-passing algorithms have been applied to a growing variety of combinatorial problems [2,8,9,10], augmenting their traditional roles in probabilistic inference [11,12,13]. The methods all operate by propagating messages between a problem's variables, causing them to iteratively adjust their own bias estimates from some initial randomized values.

The techniques produce "surveys", representing, informally, the probability that each variable should be set a certain way if we were to assemble a satisfying assignment. Thus a propagation algorithm does not output an outright solution to a SAT problem. Rather, applying a probabilistic method to SAT-solving requires two interrelated design decisions: a means of calculating surveys, and a means of using the surveys to fix the

SAT Theory: $C_1 \wedge \ldots \wedge C_8$

$$
\begin{array}{ll}
C_1 = (x_1 \vee x_2 \vee \neg x_3) & C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_4) \\
C_3 = (x_1 \vee \neg x_2 \vee \neg x_5) & C_4 = (\neg x_1 \vee x_3 \vee \neg x_4) \\
C_5 = (x_1 \vee \neg x_3 \vee x_5) & C_6 = (x_1 \vee \neg x_4 \vee x_5) \\
C_7 = (x_2 \vee x_4 \vee x_5) & C_8 = (\neg x_3 \vee x_4 \vee \neg x_5)
\end{array}
$$

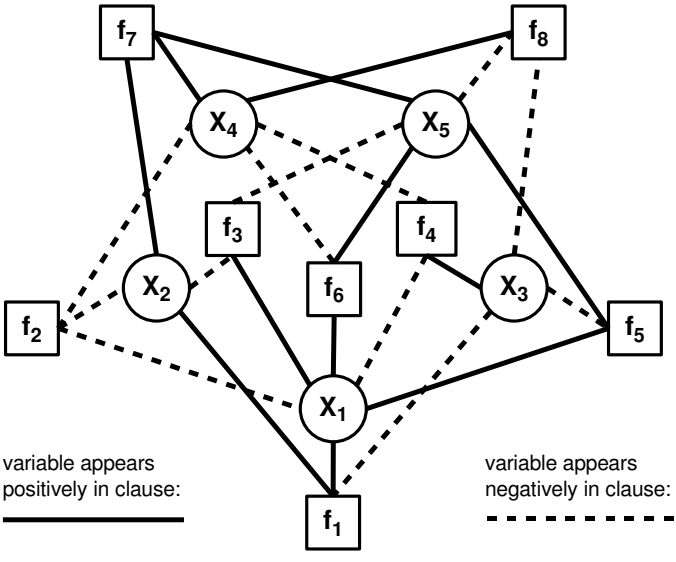


Fig. 1. Example 3-SAT Problem: as Factor Graph and as CNF Theory

x_1	x_2	x_3	x_4	x_5
$\langle$ 0,	0,	0,	0,	1 $\rangle$
$\langle$ 0,	0,	0,	1,	1 $\rangle$
$\langle$ 0,	1,	0,	0,	0 $\rangle$
$\langle$ 1,	0,	0,	0,	1 $\rangle$
$\langle$ 1,	0,	1,	1,	0 $\rangle$
$\langle$ 1,	0,	1,	1,	1 $\rangle$
$\langle$ 1,	1,	0,	0,	0 $\rangle$
$\langle$ 1,	1,	0,	0,	1 $\rangle$
$\langle$ 1,	1,	1,	0,	0 $\rangle$

$$
\begin{array}{ll}
\theta_1(+) = 6/9, & \theta_1(-) = 3/9 \\
\theta_2(+) = 4/9, & \theta_2(-) = 5/9 \\
\theta_3(+) = 3/9, & \theta_3(-) = 6/9 \\
\theta_4(+) = 3/9, & \theta_4(-) = 6/9 \\
\theta_5(+) = 5/9, & \theta_5(-) = 4/9
\end{array}
$$

Fig. 2. Solutions to Example 3-SAT Problem, and Resulting Biases

next variable within an arbitrary search framework. A reasonable strategy for the second step is to pick the variable with the most extreme bias, and set it in the direction of that bias, i.e. "succeed-first" search. But by better understanding the characteristics of

various survey techniques, we can explore more sophisticated approaches to variable and value ordering.

Note also that integrating with a solver will mean computing a new survey each time we fix a *single* variable; in other words the size of a decimation block will be one. This is a standard practice for addressing correlations between variables [3]; A single survey might report that v_1 is usually *true* within the space of solutions, and that v_2 is usually *false*, even though the two events happen simultaneously with relative infrequency. Instead of attempting to fix multiple variables at once, then, we will fix a single first and simplify the resulting problem. In subsequent surveys the other variables' biases would thus be conditioned on this first assignment.

At a high level, propagation techniques accomplish the bias estimation task by passing messages over a given SAT problem's "factor graph" representation, as depicted in Figure 1. Nodes representing variables connect to "factor" nodes representing clauses in which they appear. Edges can be distinguished, conceptually, by whether the variables appear as positive or negative literals in the clauses. Thus, for example, we see that factor f_1 realizes the disjunction $x_1 \vee x_2 \vee \neg x_3$.

The example problem has nine solutions, as listed in Figure 2. From this table we can calculate exact biases by tallying entries along each column: for instance x_1 is set positively in six out of the nine solutions, and negatively in the remaining three. (This calculation glosses the notion of variables being *constrained* positively or negatively, versus merely appearing as such; this is the distinction between BP and SP.)

Conceptually, edges carry clause-to-variable messages in one direction, and variable-to clause messages in the other. For all the techniques presented shortly, each variable is first randomly seeded with an initial bias, and informs all of its clauses by passing variable-to-clause messages along the edges. The clauses compile such reports and determine whether they are poorly supported–that is, they calculate the probability that their variables will jointly end up failing to satisfy them. From here they signal each variable as to whether they need their support by passing messages back along the edges, in the opposite direction. The variables weigh such requests, and begin a new iteration by updating and reporting their new biases.

Importantly, though, the entire factor graph framework is notional in the sense that it formalizes the derivation of our bias estimators [5], but they will never actually represent such a structure in memory, or organize a series of messages. This is crucial to the efficient implementation of such methods; in the end we will instead use a set of update rules whereby a given variable's bias is directly updated via a fixed calculation in terms of the biases of adjacent variables (that is, those with which it appears together in some clause.)

3 Definitions

Definition 1 (SAT instance). *A (CNF)* **SAT instance** *is a set C of m clauses, constraining a set V of n Boolean variables. Each clause $c \in C$ is a disjunction of literals built from the variables in V. An assignment $X \in \{0,1\}^n$ to the variables satisfies the instance if it makes at least one literal true in each clause. The sets V_c^+ and V_c^- comprise the variables appearing positively and negatively in a clause c, respectively.*

The sets C_v^+ and C_v^- comprise the clauses that contain positive and negative literals for variable v, respectively. $C_v = C_v^+ \cup C_v^-$ comprises all clauses that contain v as a whole.

Definition 2 (Bias, Survey). *For a satisfiable SAT instance $\mathcal{F}$, the* **bias distribution** *θ_v of a variable v represents the fraction of solutions to $\mathcal{F}$ wherein v appears positively or negatively. Thus it consists of a* **positive bias** *θ_v^+ and a* **negative bias** *θ_v^-, where $\theta_v^+, \theta_v^- \in [0,1]$ and $\theta_v^+ + \theta_v^- = 1$. A vector of bias distributions, one for each variable in a theory, will be called a* **survey***, denoted $\Theta(\mathcal{F})$.*

Less formally, it is useful to describe a variable as "**positively biased**" with respect to a true or estimated bias distribution. This means that under the given distribution, its positive bias exceeds its negative bias. Similarly the "**strength**" of a bias distribution indicates how much it favors one value over the other, as defined by the maximum difference between its positive or negative bias and 0.5.

4 Probabilistic Methods for Estimating Bias

In this section we present six distinct propagation methods for measuring variable bias: Belief Propagation (BP), EM Belief Propagation-Local/Global (EMBP-L and EMBP-G), Survey Propagation (SP), and EM Survey Propagation-Local/Global (EMSP-L and EMSP-G). These methods represent the space of algorithms defined by choosing between BP and SP and then employing one of them either in original form, or by applying a local- or global-consistency transformation based on the Expectation Maximization framework. The EM-based rules represent a secondary contribution of this work; they have been studied empirically without explanation [14] but have not been presented up to this point. (Complete derivations are available online [5].)

On receiving a SAT instance $\mathcal{F}$, any of the propagation methods begins by formulating an initial survey at random. For instance, the positive bias can be randomly generated, and the negative bias can be set to its complement: $\forall v, \theta_v^+ \sim \mathcal{U}[0,1]$; $\theta_v^- \leftarrow 1 - \theta_v^+$. Each algorithm proceeds to successively refine its estimates, over multiple iterations. An iteration consists of a single pass through all variables, where the bias for each variable is updated with respect to the other variables' biases, according to the characteristic rule for a method. If no variable's bias has changed between two successive iterations, the process ends with convergence; otherwise an algorithm terminates by timeout or some other parameter. EM-type methods are "convergent", or guaranteed to converge naturally, while regular BP and SP are not [3].

The six propagation methods are discussed elsewhere in greater theoretical detail than space permits here [3,2]. But for a practical understanding, they can be viewed as update rules that assign weights (ω_v^+ and ω_v^-) toward a variable's positive and negative biases–plus a third weight (ω_v^*) for the "joker bias" (described below) in the case of SP-based methods. The rules will make extensive use of the formula $\sigma(v,c)$ in Figure 3(a). In doing so they express the probability that variable v is the "sole-support" of clause c in an implicitly sampled configuration of all the variables: every other variable that appears in the clause is set unsatisfyingly. From a generative statistical perspective, the probability of this event is the product of the negative biases of all other variables

$$\sigma(v,c) \triangleq \prod_{i \in V_c^+ \setminus \{v\}} \theta_i^- \prod_{j \in V_c^- \setminus \{v\}} \theta_j^+$$

(a) $\sigma(v,c)$: v is the sole support of c.

$$\theta_v^{+\prime} \leftarrow \frac{\omega_v^+}{\omega_v^+ + \omega_v^-} \qquad \theta_v^{-\prime} \leftarrow \frac{\omega_v^-}{\omega_v^+ + \omega_v^-}$$

(b) Bias normalization for BP methods.

$$\theta_v^{+\prime} \leftarrow \frac{\omega_v^+}{\omega_v^+ + \omega_v^- + \omega_v^*} \qquad \theta_v^{-\prime} \leftarrow \frac{\omega_v^-}{\omega_v^+ + \omega_v^- + \omega_v^*} \qquad \theta_v^{*\prime} \leftarrow \frac{\omega_v^*}{\omega_v^+ + \omega_v^- + \omega_v^*}$$

(c) Bias normalization for SP methods.

Fig. 3. Formula for "sole-support"; normalizing rules for BP and SP families of bias estimators

that appear in the clause as positive literals, and the positive biases of all variables that are supposed to be negative.

The six sets of update rules produce intermediate "weight" values for each variable, by consulting the current biases of its surrounding variables. In the case of BP-based methods, each variable has two weights: one toward the positive, and one toward the negative; introducing the SP model will add a third weight toward the unconstrained state. Such weights are then normalized into proper probabilities as depicted in Figures 3(b) and 3(c), depending on whether we are using the BP or SP model. This probability constitutes a new estimated bias distribution for each variable, completing a single iteration of the algorithm. The update rules are presented below in Figures 4 and 5.

BP can be viewed at first as generating the probability that v should be positive according to the odds that one of its positive clauses is completely dependent on v for support. That is, v appears as a positive literal in some $c \in C_v^+$ for which every other positive literal i turns out negative (with probability θ_i^-), and for which every negative literal $\neg j$ turns out positive (with probability θ_j^+). This combination of unsatisfying events would be represented by the expression $\sigma(v,c)$. However, a defining characteristic of BP is its

$$\omega_v^+ = \prod_{c \in C_v^-} (1 - \sigma(v,c)) \qquad \omega_v^- = \prod_{c \in C_v^+} (1 - \sigma(v,c))$$

(a) Regular BP update rule.

$$\omega_v^+ = |C_v| - \sum_{c \in C_v^-} \sigma(v,c) \qquad \omega_v^- = |C_v| - \sum_{c \in C_v^+} \sigma(v,c)$$

(b) EMBP-L update rule.

$$\omega_v^+ = |C_v^-| \left[\prod_{c \in C_v^-} (1 - \sigma(v,c)) \right] + |C_v^+|$$

$$\omega_v^- = |C_v^+| \left[\prod_{c \in C_v^+} (1 - \sigma(v,c)) \right] + |C_v^-|$$

(c) EMBP-G update rule.

Fig. 4. Update rules for the Belief Propagation (BP) family of bias estimators

assumption that every v is the sole support of at least one clause. (Further, v cannot be the only hope of support both a positive and a negative clause simultaneously, since we are sampling from the space of *satisfying* assignments.) Thus, we should view Figure 4(a) as weighing the probability that no negative clause needs v (implying that v is positive by assumption), versus the probability that no positive clause needs v for support.

EMBP-L is the first of a set of update rules derived using the EM method for maximum-likelihood parameter estimation. This statistical technique features guaranteed convergence, but requires a bit of invention to be used as a bias estimator for constraint satisfaction problems [13,9]. Resulting rules like EMBP-L are variations on BP that calculate a milder, arithmetic average by using summation, in contrast to the harsher geometric average realized by products. This is one reflection of an EM-based method's convergence versus the non-convergence of regular BP and SP. All propagation methods can be viewed as energy minimization techniques whose successive updates form paths to local optima in the landscape of survey likelihood [3]. By taking smaller, arithmetic steps, EMBP-L (and EMBP-G) is guaranteed to proceed from its initial estimate to the nearest optimum; BP and SP take larger, geometric steps, and can therefore overshoot optima. This explains why BP and SP can explore a larger area of the space of surveys, even when initialized from the same point as EMBP-L, but it also leads to their non-convergence. Empirically, EMBP-L and EMBP-G usually converge in three or four iterations for the examined SAT instances, whereas BP and SP typically require at least ten or so, if they converge at all.

Intuitively, the equation in Figure 4(b) (additively) reduces the weight on a variable's positive bias according to the chances that it is needed by negative clauses, and vice-versa. Such reductions are taken from a smoothing constant representing the number of clauses a variable appears in overall; highly connected variables have less extreme biases than those with fewer constraints.

EMBP-G is also based on smoother, arithmetic averages, but employs a broader view than EMBP-L. While the latter is based on "local" inference, resembling generalized arc-consistency, the derivation of EMBP-G uses global consistency across all variables. In the final result, this is partly reflected by the way that Figure 5(c) weights a variable's positive bias by going through each negative clause (in multiplying by $|C_v^-|$) and *uniformly* adding the chance that *all* negative clauses are satisfied without v. In contrast, when EMBP-L iterates through the negative clauses, it considers their satisfaction on an individual basis, without regard to how the clauses' means of satisfaction might interact with one another. So local consistency is more sensitive to individual clauses in that it will subtract a different value for each clause from the total weight, instead of using the same value uniformly. At the same time, the uniform value that global consistency does apply for each constraint reflects the satisfaction of all clauses at once.

SP can be seen as a more sophisticated version of BP, specialized for SAT. To eliminate the assumption that every variable is the sole support of some clause, it introduces the possibility that a variable is not constrained at all in a given satisfying assignment. Thus, it uses the three-weight normalization equations in Figure 3(c) to calculate a three-part bias distribution for each variable: θ_v^+, θ_v^-, and θ_v^*, where '*' indicates the

$$\omega_v^+ = \prod_{c \in C_v^-} (1 - \sigma(v,c)) \cdot \rho \left[1 - \prod_{c \in C_v^+} (1 - \sigma(v,c)) \right]$$

$$\omega_v^- = \prod_{c \in C_v^+} (1 - \sigma(v,c)) \cdot \rho \left[1 - \prod_{c \in C_v^-} (1 - \sigma(v,c)) \right]$$

$$\omega_v^* = \prod_{c \in C_v} (1 - \sigma(v,c))$$

(a) Regular SP update rule.

$$\omega_v^+ = |C_v| - \sum_{c \in C_v^-} \sigma(v,c)$$

$$\omega_v^- = |C_v| - \sum_{c \in C_v^+} \sigma(v,c)$$

$$\omega_v^* = |C_v| - \sum_{c \in C_v} \sigma(v,c)$$

(b) EMSP-L update rule.

$$\omega_v^+ = |C_v^-| \prod_{c \in C_v^-} (1 - \sigma(v,c)) + |C_v^+| \left[1 - \prod_{c \in C_v^+} (1 - \sigma(v,c)) \right]$$

$$\omega_v^- = |C_v^+| \prod_{c \in C_v^+} (1 - \sigma(v,c)) + |C_v^-| \left[1 - \prod_{c \in C_v^-} (1 - \sigma(v,c)) \right]$$

$$\omega_v^* = |C_v| \prod_{c \in C_v} (1 - \sigma(v,c))$$

(c) EMSP-G update rule.

Fig. 5. Update rules for the Survey Propagation (SP) family of bias estimators

unconstrained (*a.k.a* "joker") state. Thus, in examining the weight on the positive bias in Figure 5(a), it is no longer sufficient to represent the probability that no negative clause needs v. Rather, we explicitly factor in the condition that *some* positive clause needs v, by complementing the probability that *no* positive clause needs it. This acknowledges the possibility that no negative clause needs v, but no positive clause needs it either. As seen in the equation for ω_v^*, such mass goes toward the joker state. (The parameter $\rho = 0.95$ is an optional smoothing constant explained in [15].)

For the purposes of estimating bias and finding backbones, any probability mass for θ_v^* is evenly distributed between θ_v^+ and θ_v^- when the final survey is compiled. This reflects how the event of finding a solution with v labeled as unconstrained indicates that there exists one otherwise identical solution with v set to *true*, and another with it set to *false*. So while the "joker" state plays a role between iterations in setting a variable's bias, the final result omits it for the purposes of bias estimation. (One point of future interest is to examine the prevalence of lower "joker" bias in *backdoor* variables [16].)

EMSP-L and **EMSP-G** are analogous to their BP counterparts, extended to weight the third '*' state where a variable may be unconstrained. So similarly, they can be understood as convergent versions of SP that take a locally or globally consistent view of finding a solution, respectively.

5 Practical Design Considerations

In this section we discuss the most salient design considerations for integrating the rules of the previous section within MiniSat, a backtracking DPLL solver that integrates clause learning, restarts, and pre-processing with a built-in ordering heuristic based on VSIDS [6]. In short, the efficient operation of VARSAT hinges upon the interaction of five principle design decisions: a branching strategy for ordering variables and values, a threshold for deactivating the entire bias estimation apparatus, the decimation block size for fixing variables on completing a survey, a policy for using learned clauses in surveys, and finally, the choice of bias estimation technique.

Other than the choice of branching strategy, all of these decisions seek a profitable sacrifice in accuracy in return for spending less time on computing surveys. Here it is important to note that in searching for a satisfying solution, robustness is more important than pure accuracy. Even if our bias estimator instructs us to set a variable positively when its true bias is 90% negative, there still exists some set of solutions in the resulting subproblem. Thus our system would always proceed directly to a solution without even backtracking, so long as we never set a variable to a polarity for which it has a true bias of zero.

5.1 Branching Strategy

We tested several branching strategies for using surveys as variable- and value-ordering heuristics. In addition to the "conflict-avoiding" strategy of setting the most strongly biased variable to its stronger value, we also tried to "fail-first" or streamline a problem via the "conflict-seeking" strategy of setting the strongest variable to its weaker value [17].

Additional approaches involved different ways of blending the two: for instance, one strategy might involve triggering propagations and building up a strong database of learned clauses by seeking conflicts, and then trying to find a solution within this greatly restricted search space by switching to conflict avoidance. A second motivator for seeking conflicts is unsatisfiable problems. While surveys are not well-defined for such problems, seeking conflicts can lead to shorter proofs of unsatisfiability and thus faster run-times; since we must account for the entire breadth of the search tree, we should order variables so that conflicts occur on the shallowest subtrees possible.

For mixtures of satisfiable and unsatisfiable problems like those comprising the test cases for recent SAT-Solving contests, it turns out that the single best strategy is the (presumably) most intuitive one of avoiding conflicts.

5.2 Deactivation Threshold

A second consideration when integrating with a backtracking solver is that any of the six bias estimators can be governed by a *"threshold"* parameter expressed in terms of the most strongly biased variable in a survey. For instance, if this parameter is set to 0.6, then we only persist in using surveys so long as their most strongly biased variables have a gap of size 0.6 between their positive and negative bias. As soon as we receive a survey where the strongest bias for a variable does not exceed this gap, then we deactivate the

bias estimation process and revert to using MiniSat's default variable and value ordering heuristic until the solver triggers a restart. (Note that setting this parameter to 0.0 is the same as directing the solver to never deactivate the bias estimator.)

The underlying motivation is that problems should contain a few important variables that are more constrained than the rest, and that the rest of the variables should be easy to set once these few have been assigned correctly. The aim is to detect and fix the first type of variable via strong bias estimates, and then solve the resulting subproblem using DPLL search alone with a default ordering heuristic. For various theoretical reasons, the divide between a small number of constrained variables and a large number of less important ones is thought to be of special relevance within the phase-transition region in hardness for random problems [18]. For industrial and crafted problems, the hope is that a similar distinction exists; but here the exact ratio eludes formal analysis. Most generally, surveys are highly valuable but also very expensive; they can take the majority of a solver's total runtime depending on how this parameter and others are set. So it is critical to stop computing surveys after the most important decisions have been made.

For typical SAT contest problems, we have found .9 to be a good value in combination with the other settings decided in this section. This will typically result in the execution of about one survey per thousand variables before deactivation, for large problems with tens of thousands of variables and up.[1] (Each time the solver performs a restart in solving a given problem, the bias estimation module is re-instated if it was previously deactivated.)

5.3 Decimation Block Size

Another way to mitigate the high cost of computing surveys is to use a large decimation block size, meaning that we can set a number of variables at once each time we complete a survey. The problem is then simplified via unit propagation and we compute a new survey on the resulting sub-problem.

Under preliminary investigation we have found that it is still better to use a block size of 1, though this issue is not fully resolved due to the many combinations of settings for the other parameters discussed so far. As mentioned previously, the motivation for a small decimation size is to account for correlations between variable biases. When setting a block of multiple variables, we are approximating a joint probability over their mutual configuration by the product of their individual marginal probabilities. If we instead set a single variable and compute a new survey, the resulting values are conditioned upon our previous decision. At this point it seems that the extra accuracy provided by this property is worth the greater computational cost.

5.4 Integrating Learned Clauses

Integrating learned clauses into surveys represents another balance between accuracy and runtime efficiency. On the one hand, learned clauses are all implied by a theory, and their influence is already implicitly captured by the original clauses of a problem.

[1] For the smaller problems considered in Figure 6, surveys typically wound up running over approximately a tenth of the total variables in a problem.

On the other, they may provide especially useful information about the specific area of the search space that a solver is currently exploring. There is an extra cost in runtime because update rules must now iterate through additional clauses when estimating biases; at an implementational level there is also the overhead of managing memory for registering learned clauses to be used in surveys, and for unregistering them when they are periodically purged.

The tradeoff is accomplished by a parameter that states the maximum size that a learned clause can have if it is to be used in surveys. (Shorter clauses are more valuable in the sense that they contain more information, and they are also faster to process since they appear in the update rules of fewer variables.) In practice we have found it best to integrate all learned 4-clauses and below into survey computations. For the SAT-contest problems considered, though, very few such clauses are ever learned, and the improvement over not using any learned clauses at all is small.

5.5 Bias Estimation Technique

Finally, the choice of bias estimator represents a tradeoff between runtime and various types of accuracy. This decision has the most interaction with the way the other design issues are resolved. For instance, suppose we are decimating one variable at a time and are seeking solutions by branching on the one with strongest estimated bias. Then it does us no good if our chosen estimator has excellent accuracy on the majority of the variables if it often happens that the one variable with strongest estimated bias is guessed incorrectly.

For random problems, stronger global constraints and the richer SP model make EMSP-G the best bias estimator, despite the greater cost of computing such constraints and performing three updates per variable instead of two. For industrial problems from recent SAT contests, the global constraints are still valuable but the SP model no longer seems to be worthwhile–here EMBP-G is the method of choice.

6 Empirical Performance

Here we briefly summarize the empirical performance of the completed VARSAT system. More in-depth studies of the individual bias estimators and parameter settings are detailed online [5], while performance on specific data sets will be available when the 2009 SAT contest is completed.

Figure 6 compares VARSAT's heuristic strategy with the default strategy of MiniSat 2.0, on hard random problems of increasing size. Here EMSP-G was used as the bias estimator, with deactivation threshold 0.6, to perform conflict-avoiding search with decimation block size of one and no learned clauses consulted in forming surveys.

For each problem size marked in the graph, the two solvers were run on 1000 satisfiable instances that were randomly generated with a clause-to-variable ratio of 4.11–such problems approach the hard region of the satisfiability phase transition. The average runtime on such problems is plotted in log-scale. The last two data points for default MiniSat represent lower bounds; on a percentage of the runs MiniSat timed out by failing to solve an instance within 10,800 seconds (three hours.)

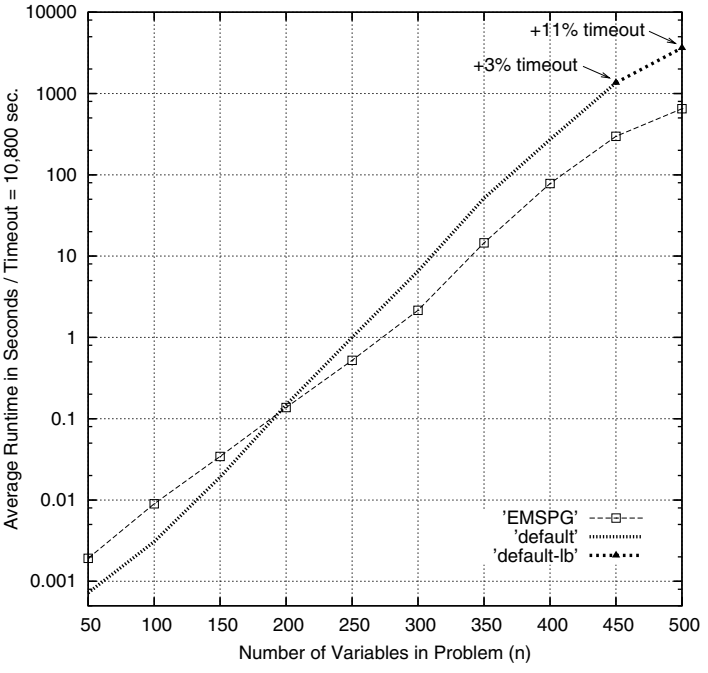


Fig. 6. Comparison on Random Problems, $n = 50 - 500$, $\alpha \equiv \frac{m}{n} = 4.11$

However, the real strength of MiniSat and other DPLL-based solvers is on industrial problems. While the results of this year's contest remain to be seen, it is possible to perform a comparison using last year's data sets. Here the best configuration of VARSAT was to use EMBP-G with a deactivation threshold of 0.9, and the other parameters remaining the same. On running the two solvers on a collection of 125 past problems with a timeout of 15 minutes, they were both able to solve 65 of them, and both failed to solve 46. Of the former, MiniSat was generally faster though both were within the time limit. Additionally, there were 4 problems that only VARSAT could solve, compared to 10 for MiniSat. Other comparisons suggest that the performance of VARSAT relative MiniSat improves given a longer timeout–on the larger problems, bias estimation takes up more than 99% of runtime, and just initializing the appropriate structures can take minutes under the current implementation.

7 Conclusions and Future Work

The main finding of this work has been that probabilistic message-passing techniques can be successfully integrated within a backtracking search framework, in order to achieve completeness and also in order to handle industrial problems. This is contingent on a number of design decisions that primarily trade accuracy for time. A second contribution is four novel bias estimators, two of which (EMBP-G and EMSP-G) have

proved to be the most useful within the resulting VARSAT solver. Integrating message-passing with DPLL combines the best properties of random-walk based methods (good performance on random problems) with backtracking methods (completeness), but in a very simple sense. In terms of raw performance, VARSAT is comparable but still generally slower than MiniSat on non-random problems.

However, the overall framework is new and not close to fully explored. The parameters discussed in Section 5 produce a complex of combinations and interactions that should be resolved more systematically according to the type of problem being solved. Statistical analysis provides one automated means for doing so [19,20]. Unsatisfiable problems are such a type of especial future interest; while the scheme presented here works reasonably well on satisfiable and unsatisfiable problem instances alike, the actual semantics of bias over the latter type of instance eludes easy definition. At any rate it appears likely that a specialized set of parameter settings will improve future efforts to apply VARSAT to unsatisfiable problems.

Another task is to speed up the bias estimation process by better optimizing its code at an implementational level, possibly to include porting it to alternative hardware [21]. Algorithmically, an additional possibility is to use only a portion of a variable's clauses in estimating its bias.

There are interesting abstract similarities with other problem-solving methodologies for constraint satisfaction. Bias measures the probability of a variable setting given satisfaction, while many local search methods maximize the probability of satisfaction given a certain variable setting [22]. Thus, the two targets are directly proportional via Bayes' Rule and techniques for one can be applied to the other. Another line of similar research calculates exact solution counts for individual constraints as a means of ordering variables and values [23]. Fundamentally, such an approach represents an exact and localized version of the approximate and interlinked techniques studied here. Finally, another means to making incomplete search complete is to define a gradient function for local search and register local minima by adding successive constraints until the resulting problem is convex [24]. The probabilistic (marginal computation) methods presented here are not strictly related to local search, but they are very similar to gradient-based (MAP-computation) methods that follow the same framework [7].

Future applications of bias estimation include query answering and model counting. In the case of model counting, one detail omitted from discussion is that the normalization value $\omega_v^+ + \omega_v^-$ in Figure 3(b) (in fact, the log-partition function of a specific Markov Random Field) is proportional to the number of solutions for a given problem.

References

1. Dechter, R., Kask, K., Mateescu, R.: Iterative join-graph propagation. In: Proc. of 18th International Conference on Uncertainty in Artificial Intelligence (UAI 2002), Edmonton, Canada, pp. 128–136 (2002)
2. Braunstein, A., Mezard, M., Zecchina, R.: Survey propagation: An algorithm for satisfiability. Random Structures and Algorithms 27, 201–226 (2005)
3. Hsu, E., McIlraith, S.: Characterizing Propagation Methods for Boolean Satisfiability. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 325–338. Springer, Heidelberg (2006)

4. Achlioptas, D., Ricci-Tersenghi, F.: Random formulas have frozen variables. SIAM Journal of Computing (to appear)
5. Hsu, E.I.: VARSAT SAT-Solver homepage (2008), http://www.cs.toronto.edu/~eihsu/VARSAT/
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
7. Jordan, M., Ghahramani, Z., Jaakkola, T., Saul, L.: An introduction to variational methods for graphical models. In: Jordan, M. (ed.) Learning in Graphical Models. MIT Press, Cambridge (1998)
8. Kask, K., Dechter, R., Gogate, V.: Counting-based look-ahead schemes for constraint satisfaction. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 317–331. Springer, Heidelberg (2004)
9. Hsu, E., Kitching, M., Bacchus, F., McIlraith, S.: Using EM to find likely assignments for solving CSP's. In: Proc. of 22nd Conference on Artificial Intelligence (AAAI 2007), Vancouver, Canada (2007)
10. Kroc, L., Sabharwal, A., Selman, B.: Survey propagation revisited. In: Proc. of 23rd International Conference on Uncertainty in Artificial Intelligence (UAI 2007), Vancouver, Canada (2007)
11. Pearl, J.: Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann, San Mateo (1988)
12. Kschischang, F.R., Frey, B.J., Loeliger, H.A.: Factor graphs and the sum-product algorithm. IEEE Transactions on Information Theory 47(2) (2001)
13. Dempster, A., Laird, N., Rubin, D.: Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statistical Society 39(1), 1–39 (1977)
14. Hsu, E., Muise, C., Beck, J.C., McIlraith, S.: Probabilistically estimating backbones and variable bias: Experimental overview. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202. Springer, Heidelberg (2008)
15. Maneva, E., Mossel, E., Wainwright, M.: A new look at survey propagation and its generalizations. Journal of the ACM 54(4), 2–41 (2007)
16. Williams, R., Gomes, C., Selman, B.: Backdoors to typical case complexity. In: Proc. of 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico (2003)
17. Beck, J.C., Prosser, P., Wallace, R.J.: Trying again to fail-first. In: Faltings, B.V., Petcu, A., Fages, F., Rossi, F. (eds.) CSCLP 2004. LNCS (LNAI), vol. 3419, pp. 41–55. Springer, Heidelberg (2005)
18. Braunstein, A., Zecchina, R.: Survey propagation as local equilibrium equations. Journal of Statistical Mechanics: Theory and Experiments PO6007 (2004)
19. Wallace, R.J.: Factor analytic studies of CSP heuristics. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 712–726. Springer, Heidelberg (2005)
20. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research 32, 565–606 (2008)
21. Manolios, P., Zhang, Y.: Implementing survey propagation on graphics processing units. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 311–324. Springer, Heidelberg (2006)
22. Zhang, W.: Configuration landscape analysis and backbone guided local search. Part I: Satisfiability and maximum satisfiability. Artificial Intelligence 158(1), 1–26 (2004)
23. Zanarini, A., Pesant, G.: Solution counting algorithms for constraint-centered search heuristics. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 743–757. Springer, Heidelberg (2007)
24. Fang, H., Ruml, W.: Complete local search for propositional satisfiability. In: Proc. of 19th National Conference on Artificial Intelligence (AAAI 2004), San Jose, CA (2004)

Resolution and Expressiveness of Subclasses of Quantified Boolean Formulas and Circuits

Hans Kleine Büning[1], Xishun Zhao[2], and Uwe Bubeck[1]

[1] Universität Paderborn, Germany
kbcsl@upb.de, bubeck@upb.de
[2] Sun Yat-sen University Guangzhou, PR China
hsszxs@mail.sysu.edu.cn

Abstract. We present an extension of Q-Unit resolution for formulas that are not completely in clausal form. This *b-unit resolution* is applied to different classes of quantified Boolean formulas in which the existential and universal variables satisfy the Horn property. These formulas are transformed into propositional equivalents consisting of only polynomially many subformulas. We obtain compact encodings as Boolean circuits and show that both representations have the same expressive power.

1 Introduction

Recently, there has been growing interest [7, 8] in non-clausal or structural quantified Boolean formulas (QBF or QBF* if free variables are allowed). Accordingly, we present an extension of Q-Unit resolution, denoted *b-unit resolution*, for formulas that are not completely in clausal form. We relate the idea to Boolean circuits which have the ability to use intermediate results in multiple places by fan-out, so that we avoid copying of resolvents in our b-unit resolution.

We begin with some definitions. A QBF* formula Φ is *satisfiable* if there is a truth assignment v to the free variables $\mathbf{z}$ such that Φ is true after substituting the truth values v for the free variables. For $\Phi \in$QCNF*, we write $\Phi = Q \bigwedge_i (\phi_i^b \vee \phi_i^f)$, where the *b-part* ϕ_i^b is a clause over bound variables and the *f-part* ϕ_i^f is a clause over free variables. QHORN* is the set of quantified Horn formulas with free variables, i.e. formulas $Q\phi$ where ϕ is a Horn formula. QHORN^b is the set of formulas where each b-part ϕ_i^b is a Horn clause and ϕ_i^f an arbitrary clause over free variables. QHORN$^+$ (QHORN$^-$) is the subset of QHORN^b for which the f-part of each clause is a disjunction of positive (negated) variables.

A circuit is a DAG with one outgoing edge and multiple input nodes labeled with Boolean variables. The other nodes are AND-, OR-, and NOT-gates that each have two (AND and OR) or one (NOT) incoming edges. The *fan-out* of a circuit is the maximum number of outgoing edges of the AND- and OR-gates. We can transform in linear time any circuit into *standard form*, where the inner nodes are only AND- and OR-gates and the inputs are variables x and/or negated variables $\neg x$. Subsequently, we focus on the class C of circuits in standard form.

A *monotone* propositional formula contains no negations. *Anti-monotone* formulas are negated monotone formulas. Analogously, *monotone* circuits $\mathcal{C}_{mon}$

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 391–397, 2009.

have only non-negated variables as inputs. *Anti-monotone* circuits $\mathcal{C}_{anti-mon}$ have only negated inputs $\neg x$. Suppose we have Horn clauses $(\alpha_1 \rightarrow x)$, ..., $(\alpha_n \rightarrow x)$. We can combine these into $((\alpha_1 \vee \ldots \vee \alpha_n) \rightarrow x)$, which is not a Horn clause, but $(\alpha_1 \vee \ldots \vee \alpha_n)$ is monotone and thus equivalent to a monotone circuit. More generally, we introduce *$\mathcal{C}$-Horn clauses* $(c \rightarrow z)$, where c is a monotone propositional formula and z a variable. $(c \rightarrow z)$ can be represented as a circuit $(z \vee \neg c)$ with monotone c. This non-standard circuit can be transformed into $(z \vee c')$ in standard form, where $c' \approx \neg c$ and c' is anti-monotone. A conjunction $\bigwedge_i (c_i \rightarrow z_i)$ of circuits that represent $\mathcal{C}$-Horn clauses is called a $\mathcal{C}_{Horn}$ circuit.

For $i = 1, 2$, let $\Phi_i(\mathbf{z})$ be a propositional formula over variables $\mathbf{z}$, a QBF* formula with free variables $\mathbf{z}$, or a circuit with input variables $\mathbf{z}$. Then Φ_1 and Φ_2 are *equivalent* ($\Phi_1 \approx \Phi_2$) if and only if for every truth assignment v over $\mathbf{z}$ we have $v(\Phi_1) = v(\Phi_2)$. The *size* $|c|$ of a circuit c is the number of gates. For a formula Φ, $|\Phi|$ is its *length*. The usual definition is to count the number of occurrences of variables, including the prefix. Without multiple negations ($\neg\neg x$), this differs only by a constant factor from the number of operators.

Definition 1. *For classes A, B of propositional or QBF* formulas or circuits, we let $A \leq_p^r B$ if and only if there is a polynomial q such that for any $\alpha \in A$ there is $\beta \in B$ with $\alpha \approx \beta$ and $|\beta| \leq q(|\alpha|)$. $A =_p^r B$ if $A \leq_p^r B$ and $B \leq_p^r A$.*

2 Extensions of Unit Resolution

It is well known that unit resolution is complete for Horn formulas. *Q-Unit resolution* [5] extends the idea to QCNF* by resolving on free and existential literals where one of the parent clauses has exactly one such literal. This is correct and refutation-complete [5] for formulas $\Phi = Q(\alpha_1 \wedge \ldots \wedge \alpha_m)$ with free variables $\mathbf{z}$ in which for every clause α_i the existential and free literals form a Horn clause, i.e. after eliminating all universals the clause is in HORN. Such formulas are called *quantified extended Horn* (QEHORN*). The satisfiability problem for this class has been shown to be PSPACE-complete in general and coNP-complete for a fixed number of prefix alternations $(\forall\exists)^k$, $k \geq 1$ [4]. There exist QEHORN formulas for which every resolution refutation requires exponentially many steps [5].

We now present an extension of Q-Unit resolution for formulas that are not completely in clausal form. Let $\Phi = Qv_1 \ldots Qv_n \bigwedge_i (\phi_i^b \vee \alpha_i)$ be in QBF*, where ϕ_i^b is a Horn clause over bound variables and α_i an arbitrary propositional formula over free variables. Then Φ is not in QHORN^b if α_i is not a disjunction of literals. But ϕ_i^b is a Horn formula on which we can apply unit resolution.

Definition 2. *We say that $(L \vee \alpha)$ is a* b-unit clause *if L is a literal over an existentially quantified variable and α is a formula over free variables.*

Let $(L \vee \alpha_1)$ be a b-unit clause, $(\neg L \vee \beta)$ a Horn clause over bound variables, and α_2 a formula over free variables. Then we define b-unit resolution *as*

$$(L \vee \alpha_1), (\neg L \vee \beta \vee \alpha_2) \vdash_{\overset{1}{b\text{-}Unit\text{-}Res}} (\beta \vee \alpha_1 \vee \alpha_2) \ .$$

Q-Unit resolution is only refutation complete in combination with universal reduction, that is, the removal of universals that do not dominate any existential in the same clause. We also have to be careful not to resolve clauses with tautological universals. Such blockings usually require detours in resolution derivations, making them longer. While $\exists$-unit clauses in Q-Unit resolution may have an arbitrary number of universals, our definition of b-unit resolution avoids these difficulties by requiring that b-unit clauses have exactly one bounded literal which is existential. This is justified by the following result on QHORN^b formulas.

It has been shown in [3] that any QHORN* formula Φ can be transformed into an equivalent $\Phi' \in \exists$HORN*, such that the length of Φ' and the time for executing the transformation are less than quadratic in $|\Phi|$. That proves QHORN$^* =_p^r \exists$HORN*. A careful analysis of the transformation shows that the free parts of the clauses remain unchanged. Thus, $\exists$HORN$^- =_p^r$ QHORN$^-$. For $\Phi \in$ QHORN$^+$, we substitute the positive occurrences of free variables by their complements. Then the formula is in QHORN$^-$ and has an equivalent formula in $\exists$HORN$^-$ of at most quadratic length. We reverse the substitution and obtain a formula in $\exists$HORN$^+$ equivalent to Φ and with length at most quadratic in $|\Phi|$.

For QHORN$^b \leq_p^r \exists$HORN^b, let $\Phi(\mathbf{z}) = Q\ (\bigwedge_{1\leq i\leq m}(\phi_i^b \vee \phi_i^f))$ be a QHORN^b formula. We introduce for each clause a new variable w_i and replace ϕ_i^f with $\neg w_i$. We get the QHORN$^-$ formula $\Phi(\mathbf{w}) = Q \bigwedge_{1\leq i\leq m}(\phi_i^b \vee \neg w_i)$. Because of $\exists$HORN$^- =_p^r$ QHORN$^-$, there is an $\exists$HORN$^-$ formula $\Phi'(\mathbf{w}) = \exists \mathbf{y} \bigwedge_j(\varphi_j^b \vee \varphi_j^f)$ of quadratic length with $\Phi(\mathbf{w}) \approx \Phi'(\mathbf{w})$. For $1 \leq i \leq m$, we now replace $\neg w_i$ with ϕ_i^f and obtain for $(\varphi_j^b \vee \neg w_{i_1} \vee \cdots \vee \neg w_{i_r})$ the clause $(\varphi_j^b \vee \phi_{i_1}^f \vee \cdots \vee \phi_{i_r}^f)$. The result is equivalent to Φ and is in $\exists$HORN^b with length polynomial in $|\Phi|$.

Lemma 1. $\exists$HORN$^\circ =_p^r$ QHORN$^\circ$ *for* $\circ \in \{*, b, +, -\}$ *by polynomial-time transformations.*

Each step of b-unit resolution can be simulated by a series of regular Q-Unit resolution steps. Let $Q\phi = Q(\phi' \wedge (L \vee \alpha_1) \wedge (\neg L \vee \beta \vee \alpha_2))$ be the formula which contains the two extended clauses to be resolved. Then we transform $(L \vee \alpha_1)$ into an equivalent conjunction of clauses $(L \vee \alpha_{1,1}) \wedge ... \wedge (L \vee \alpha_{1,r})$, and similarly $(\neg L \vee \beta \vee \alpha_2)$ into $(\neg L \vee \beta \vee \alpha_{2,1}) \wedge ... \wedge (\neg L \vee \beta \vee \alpha_{2,s})$. Now we perform all possible Q-Unit resolutions over L. The definition of Q-Unit resolution implies $Q\psi \approx Q(\psi \wedge \sigma)$ for every resolvent σ [6]. In our case, it follows that $Q\phi \approx Q(\phi \bigwedge_{i,j}(\beta \vee \alpha_{1,i} \vee \alpha_{2,j}))$. Since all resolvents contain β, we pull it out $\bigwedge_{i,j}(\beta \vee \alpha_{1,i} \vee \alpha_{2,j}) \approx \beta \vee \bigwedge_{i,j}(\alpha_{1,i} \vee \alpha_{2,j})$. Now we can reverse the CNF transformation of α_1 and α_2: $\beta \vee \bigwedge_{i,j}(\alpha_{1,i} \vee \alpha_{2,j}) \approx \beta \vee \bigwedge_i(\alpha_{1,i} \vee \bigwedge_j \alpha_{2,j}) \approx \beta \vee \bigwedge_i(\alpha_{1,i} \vee \alpha_2) \approx \beta \vee \alpha_1 \vee \alpha_2$, which is the b-unit resolvent as defined above.

Proposition 1. *Let $\Phi = Q\phi$ be a QBF* formula, and let σ be a b-unit resolvent $\Phi \vdash^1_{\text{b-Unit-Res}} \sigma$. Then we have $Q\phi \approx Q(\phi \wedge \sigma)$.*

So, b-unit resolution is a way to perform multiple unit resolution steps at once. We attempt to make even more use of this capability by actively combining multiple b-unit clauses with the same bound variable into a larger b-unit clause.

Definition 3. *Let* $\varphi = \{F_{1,1} \to x_1 ... F_{r_1,1} \to x_1, ..., F_{1,m} \to x_m ... F_{r_m,m} \to x_m\}$ *be a set of b-unit clauses.* $F_{i,j}$ *contains only free variables, and* x_j *is bound.*

We define $cmb(\varphi) := \{(F_{1,1} \vee \ldots \vee F_{r_1,1}) \to x_1, \ldots, (F_{1,m} \vee \ldots \vee F_{r_m,m}) \to x_m\}$.

3 Structure of Resolvents and Circuits

We now want to derive by b-unit resolution from a given ∃HORN* formula a quantifier-free formula $(F \to z)$ where F is a monotone propositional formula. While F may have exponential size, we show that it essentially consists of only at most quadratically many different subformulas, because it can be derived by a quadratic number of b-unit resolution steps, where each resolvent can be represented by a linear-size circuit. By fan-out greater than 1, the substitution of one resolvent into another one can be performed without copying. The ability of b-unit resolution to work on non-CNF avoids subformulas being torn apart by repeated CNF transformation. The following example illustrates the idea: Let $\Phi = \exists x_1 \exists x_2 \exists x_3 \exists y\, (\neg y \vee z) \wedge (a \to x_1) \wedge (b \to x_1) \wedge (c \wedge x_1 \to y) \wedge (d \wedge x_1 \to x_3)$. Φ contains the b-units $T(0) := \{(a \to x_1), (b \to x_1)\}$. We combine these into $G(0) := \{(a \vee b) \to x_1\}$. Then we resolve the units in $G(0)$ with the clauses in Φ by b-unit resolution and get $T(1) := \{(c \wedge (a \vee b) \to y), (d \wedge (a \vee b) \to x_3)\} \cup T(0)$. The combined b-units are $G(1) := \{((a \vee b) \to x_1), (c \wedge (a \vee b) \to y), (d \wedge (a \vee b) \to x_3)\}$. Further propagation does not lead to new combined b-units. Finally, we resolve on the clause $(\neg y \vee z)$ with negative b-part and get $T^f = ((c \wedge (a \vee b)) \to z) \approx \Phi$. This leads to the algorithm in Listing 1.

Listing 1. ∃HORN* to C_{Horn} Transformation

Input $\Phi(\mathbf{z}) = \exists \mathbf{x} \phi \in \exists$HORN* with free variables $\mathbf{z} = z_0, \ldots, z_m$
and n clauses, each containing a bound variable.
$\Phi^b = \exists \mathbf{x} \phi^b$ is unsatisfiable, ϕ contains exactly one clause
$\phi_1 = (B_1 \to z_0)$ whose bound part is a negative clause;

$T(0)$:= $\{(F \to x) \in \phi \mid x$ bound, F has only (positive) free vars$\}$;
$G(0)$:= $cmb(T(0))$;
for each $(F \to x) \in G(0)$
build a monotone circuit $c_x(0) \approx F$ with output labeled x;
for $(k$ = 0 to $n)$ {
$T(k+1)$:= $\{\psi[x_1/F_1, \ldots, x_r/F_r] \to x \mid (\psi \to x) \in \phi,\ (F_i \to x_i) \in G(k),$
$x_1, \ldots, x_r$ are the bound variables in ψ, x is bound$\}$
for each $(\psi' \to x) \in T(k+1)$
build a monotone circuit $c_{\psi'}(k+1) \approx \psi' = \psi[x_1/F_1, \ldots, x_r/F_r]$
with output labeled ψ' by reusing the circuits $c_{x_i}(k)$;
$G(k+1)$:= $cmb(G(k) \cup T(k+1))$;
for each $(F \to x) \in G(k+1)$
build a monotone circuit $c_x(k+1) \approx F$ with output labeled x
by reusing the circuits $c_{x_i}(k)$ and $c_{\psi'_j}(k+1)$;
}
T^f := $(B_1[x_1/F_1, \ldots, x_r/F_r] \to z_0)$

where $x_1, \ldots, x_r$ are the bound variables in the distinguished clause $(B_1 \to z_0)$ and $(F_i \to x_i) \in G(n+1)$;
combine circuits $c_{x_1}(n+1), \cdots, c_{x_r}(n+1)$ by AND–gates into a monotone circuit $c_\Phi \approx B_1[x_1/F_1, \ldots, x_r/F_r]$;

Output C_{Horn} circuit $c \approx z_0 \vee \neg c_\Phi$. It follows that $c \approx \Phi(\mathbf{z})$.

The algorithm requires some initial transformations. Each $\Phi = Q \bigwedge_i (\phi_i^b \vee \phi_i^f)$ can be converted in polynomial time into an equivalent formula such that every f-part contains at most one literal. Let $\phi_i^f = (\alpha \vee \beta)$ and $\phi_i^b = (\varphi_1 \vee \varphi_2)$ where φ_1 and φ_2 contain the negative and the positive literals, respectively. Then we introduce a new bound variable y and replace ϕ_i with $(\varphi_2 \vee \neg y \vee \alpha)$ and $(\varphi_1 \vee y \vee \beta)$. The monotone or anti-monotone structure of the f-parts and the Horn structure of the b-parts is preserved. So we assume that the clauses in QHORN* (QHORN$^+$, QHORN$^-$, QHORN^b) formulas contain at most one free literal such that the complete clause is a Horn clause (the f-part is a positive literal, the f-part is a negative literal, the f-part is an arbitrary free literal). Clauses $\phi_j = \phi_j^f$ can be shifted before the prefix, such that $\Phi \approx \phi_j \wedge Q \bigwedge_{i \neq j} \phi_i$. We therefore focus on formulas in which every clause contains a bound variable. We also require that every bound variable has at least one positive and one negative occurrence.

We can decide in linear time whether the conjunction $\Phi^b := \bigwedge_i \phi_i^b$ of all b-parts is satisfiable, because Horn satisfiability is solvable in linear time. If Φ^b is indeed satisfiable, $\Phi(\mathbf{z})$ is true for any truth assignment to the free variables and can be replaced by $(z \vee \neg z)$. Hence, we assume that Φ^b is unsatisfiable. Since any minimal unsatisfiable Horn formula contains exactly one negative clause in addition to the mixed clauses, we divide the formula into multiple subformulas that each contain a single negative clause ϕ_i^b. Suppose Φ has the negative b-parts $\phi_1^b, \ldots, \phi_r^b$. Let $\phi' := \phi - \{\phi_1, \ldots, \phi_r\}$. Then $\exists \mathbf{x} \phi \approx \exists \mathbf{x}(\phi' \wedge \phi_1) \wedge \ldots \wedge \exists \mathbf{x}(\phi' \wedge \phi_r)$.

The clauses ϕ_i have the form $\phi_i = (\neg x_{j_1} \vee \ldots \vee \neg x_{j_s} \vee \neg z_{k_1} \vee \ldots \vee \neg z_{k_t} \vee z_0)$ for free variables $z_{k_1}, \ldots, z_{k_t}, z_0$. W.l.o.g., we assume $\phi_i = (\neg x_{j_1} \vee \ldots \vee \neg x_{j_s} \vee z_0)$ without negative free variables. If that were not the case for some ϕ_i, we could split it into $(\neg x_{j_1} \vee \ldots \vee \neg x_{j_s} \vee \neg z_{k_1} \vee \ldots \vee \neg z_{k_t} \vee \tilde{x})$ and $(\neg \tilde{x} \vee z_0)$ by introducing a new bound variable $\tilde{x}$. Now the only clause with negative b-part is $(\neg \tilde{x} \vee z_0)$.

From Listing 1, it is clear that the size of the circuit $c_\Phi \to z_0$ is polynomial in $|\phi|$, because the number of b-units in $T(i)$ and $G(i)$, $0 \leq i \leq n+1$, is each bounded by the number of clauses in Φ, and each circuit that represents one of these b-units has linear size due to the reusing of existing circuits. The equivalence of Φ and T^f follows in the direction from left to right immediately from Proposition 1. In the other direction, it is possible to show that for truth assignments V with $V \models T^f$, V implies enough left hand sides of b-unit clauses $(F_i \to x_i) \in G(n+1)$ such that ϕ is satisfied by V and $x_i = 1$ for these x_i.

Theorem 1. *Let $\Phi = \exists \mathbf{x} \phi$ be the input to the transformation in Listing 1. In polynomial time, the algorithm computes T^f with $T^f \approx \exists \mathbf{x} \phi$. T^f can be represented by a C_{Horn} circuit of polynomial size, and thus, $\exists HORN^* \leq_p^r \mathcal{C}_{Horn}$.*

Any $\Phi \in \exists\text{HORN}^-$ is in $\exists\text{HORN}^*$ without positive free literals. Then the algorithm produces a disjunction of anti-monotone circuits $c_1, \ldots, c_r$. The disjunction of anti-monotone circuits is again anti-monotone, so $\exists\text{HORN}^- \leq_p^r \mathcal{C}_{anti-mon}$.

For $\Phi \in \exists\text{HORN}^+$, we replace the free literals with their complements and obtain a formula in $\exists\text{HORN}^-$ and then an equivalent anti-monotone circuit. We reverse the substitution and obtain a monotone circuit. Then $\exists\text{HORN}^+ \leq_p^r \mathcal{C}_{mon}$.

For $\Phi \in \exists\text{HORN}^b$, the f-parts ϕ_i^f are arbitrary clauses over free variables. For each ϕ_i, we choose a new variable w_i that replaces ϕ_i^f. The result is in $\exists\text{HORN}^+$, and there is an equivalent monotone circuit c. For each ϕ_i^f, we build an equivalent circuit c_i with output y_i and connect it to the input w_i of c. The new circuit is equivalent to Φ, and its size is polynomial in $|\Phi|$. Thus, $\exists\text{HORN}^b \leq_p^r \mathcal{C}$.

The well-known transformation of circuits to formulas [1, 2] produces $\exists\text{HORN}^b$ formulas. A close look at these for monotone, anti-monotone and C_{Horn} circuits shows that the above polynomial-size relations also hold in the other direction.

Theorem 2. *(Quantified Horn Formulas and Circuits)*
By polynomial-time transformations, we have:

1. $\text{QHORN}^+ =_p^r \exists\text{HORN}^+ =_p^r \mathcal{C}_{mon}$
2. $\text{QHORN}^- =_p^r \exists\text{HORN}^- =_p^r \mathcal{C}_{anti-mon}$
3. $\text{QHORN}^* =_p^r \exists\text{HORN}^* =_p^r \mathcal{C}_{Horn}$
4. $\text{QHORN}^b =_p^r \exists\text{HORN}^b =_p^r \mathcal{C}$

The latter constitutes an alternative proof to an earlier result $\exists\text{HORN}^b =_p^r \mathcal{C}$ by Anderaa and Börger [1], which is based on the fact that Horn satisfiability is solvable by a polynomial-time deterministic Turing machine, which in turn can be encoded by a uniform family of polynomial-size circuits.

4 Conclusion

By developing b-unit resolution for formulas that are not completely in clausal form, we have shown that various classes of quantified Boolean formulas in which the bound variables satisfy the Horn property can be transformed into quantifier-free formulas consisting of only polynomially many subformulas. These have compact encodings as circuits, and vice versa, which shows that both representations have the same expressive power, even if universal quantifiers are allowed.

References

[1] Anderaa, S., Börger, E.: The Equivalence of Horn and Network Complexity for Boolean Functions. Acta Informatica 15, 303–307 (1981)
[2] Bauer, M., Brand, D., Fischer, M., Meyer, A., Paterson, M.: A Note on Disjunctive Form Tautologies. SIGACT News 5(2), 17–20 (1973)
[3] Bubeck, U., Kleine Büning, H.: Models and Quantifier Elimination for Quantified Horn Formulas. Discrete Applied Mathematics 156(10), 1606–1622 (2008)

[4] Flögel, A., Karpinski, M., Kleine Büning, H.: Subclasses of Quantified Boolean Formulas. In: Schönfeld, W., Börger, E., Kleine Büning, H., Richter, M.M. (eds.) CSL 1990. LNCS, vol. 533, pp. 145–155. Springer, Heidelberg (1991)
[5] Flögel, A., Karpinski, M., Kleine Büning, H.: Resolution for Quantified Boolean Formulas. Information and Computation 117(1), 12–18 (1995)
[6] Karpinski, M., Kleine Büning, H., Schmitt, P.: On the computational complexity of quantified Horn clauses. In: CSL 1987. LNCS, vol. 329, pp. 129–137. Springer, Heidelberg (1988)
[7] Lonsing, F., Biere, A.: Nenofex: Expanding NNF for QBF Solving. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 196–210. Springer, Heidelberg (2008)
[8] Sabharwal, A., Ansotegui, C., Gomes, C., Hart, J., Selman, B.: QBF Modeling: Exploiting Player Symmetry for Simplicity and Efficiency. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 382–395. Springer, Heidelberg (2006)

A Compact Representation for Syntactic Dependencies in QBFs

Florian Lonsing and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria
http://fmv.jku.at/

Abstract. Different quantifier types in Quantified Boolean Formulae (QBF) introduce variable dependencies which have to be taken into consideration when deciding satisfiability of a QBF. In this work, we focus on dependencies based on syntactically connected variables. We generalize our previous ideas for efficiently representing dependency sets of universal variables to existential ones. We obtain a dependency graph which is applicable to arbitrary QBF solvers. The core part of our work is the formulation and correctness proof of a static and compact, tree-shaped connection relation over equivalence classes of existential variables. In practice, this relation is constructed once from a given QBF and allows to share connection information among all variables. We report on practical aspects and demonstrate the effectiveness of our approach in experiments on structured formulae from QBF competitions. Further, we show by example that the common approach of quantifier scope analysis is not optimal among syntactic methods for dependency computation.

1 Introduction

In the logic of Quantified Boolean Formulae (QBF), variables can be existentially or universally quantified. This extends propositional logic (SAT), where all variables are existentially quantified, and renders the decision problem of QBF PSPACE-complete [26]. Whereas QBF is not more expressive than SAT, relevant problems from formal verification [6,11,19] often can be encoded more compactly in QBF than in SAT.

The two quantifier types in QBF introduce dependencies between differently quantified variables. For example, if (the value of) an existential variable y depends on (the value of) a universal variable x, then a search-based QBF solver must not assign y before x to ensure soundness.

Example 1. In the satisfiable QBF $\forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee y)$, y depends on x. If erroneously y is assigned before x then satisfiability can not be concluded.

Dependencies limit the solver's freedom to assign variables and thus influence its performance negatively and complicate the integration of unit propagation and learning as reported in [16,17,18,21,28]. The problem of determining smallest

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 398–411, 2009.

possible dependency sets is therefore closely related to the practical applicability of QBF solvers. This also applies to memory-bound solvers which eliminate variables, for example by expansion [7,8] or skolemization [5,20].

Identifying dependencies in QBFs has been addressed in various ways in previous approaches. Most QBF solvers process formulae in prenex conjunctive normal form (PCNF), where all quantifiers occur in the quantifier prefix and the quantifier-free part of the formula is in CNF. For example, in search-based solvers like [10,14,28], dependencies are given by the total linear quantifier ordering in the prefix. Strategies for converting QBFs into PCNF were suggested in [12] to produce optimal prefixes with respect to the number of quantifier alternations. As a more powerful approach, *mini-scoping* was used in [2] to minimize quantifier scopes by shifting quantifiers from the prefix into the formula. Mini-scoping results in a tree shaped dependency relation, which follows the formula structure.

By a similar approach in [4], syntactic quantifier trees were extracted from a PCNF to be used instead of the linear prefix. In expansion-based solvers like [7,8], dependencies are identified by variable connections. A partial quantifier ordering was derived in [18] by analyzing the quantifier scope structure in non-PCNF formulae prior to conversion into PCNF. Again this results in a tree-shaped prefix which leaves more freedom for choosing decision variables. The same method can implicitly be applied in non-PCNF solvers [13]. All of these approaches mentioned so far are based on syntactic analysis of the QBF.

Informally, y depends on x in a QBF if reordering the quantifiers of x and y in the prefix changes satisfiability. For example, the formula in Ex. 1 becomes unsatisfiable under the prefix $\exists y \forall x$. Dependencies were formalized in [25] in terms of *dependency schemes*. A dependency scheme for a QBF is a binary relation D on the set of variables where $(x, y) \in D$ if y depends on x. In practice, D must be computed according to some strategy which influences the quality of D. Trivially D could be defined to correspond to the prefix: $(x, y) \in D$ if y occurs to the right of x in the prefix and is quantified differently. Such trivial dependency scheme is usually too restrictive. The goal is to minimize dependencies.

Since the problem of computing the optimal, that is the smallest, dependency scheme is PSPACE-hard [25], a trade-off has to be found between efficiency (polynomial time computation) and optimality (non-optimal over-approximation). In this work we focus on dependency computation for QBFs in PCNF by the *standard dependency scheme* D^{std} defined in [25], which is another syntactic approach based on variable connections [7,8]. As we show, D^{std} can be efficiently represented as a compact graph. This first result gives a structural characterization of the standard dependency scheme. We then show how this graph can be constructed and give experimental results.

Before elaborating our ideas, we review dependency computation by mini-scoping [2,4] and point out two drawbacks compared to our approach using D^{std}. While considering QBFs in PCNF, we argue that our results can be extended to QBFs with tree-shaped prefixes. Thus they are also applicable to solvers using quantifier scope analysis [4,13,18]. Again, using a less restrictive (that is smaller) dependency relation provides more flexibility.

1.1 Motivation

Mini-scoping was applied in various contexts as a syntactic method for dependency computation [2,4,5,7,8,12]. By rule $(Qx.\ (\phi \wedge \psi)) \equiv (Qx.\ \phi) \wedge \psi$ where $x \notin Var(\psi), Q \in \{\forall, \exists\}$, quantifiers are shifted from the prefix into the formula. Their scopes are reduced to a subset of clauses. This produces a syntactic quantifier tree (parse tree) similar to [4]. For a quantifier tree and a variable x, all differently quantified descendants of x are regarded as depending on x.

Example 2. Consider the QBF $\exists a, b \forall x, y \exists c, d.\ (a \vee x \vee c) \wedge (a \vee b) \wedge (b \vee d) \wedge (y \vee d)$. Minimizing $\exists c$, $\exists d$, $\forall x$ and $\forall y$ yields $\exists a, b.\ (\forall x \exists c.\ (a \vee x \vee c)) \wedge (a \vee b) \wedge (\forall y \exists d.\ (b \vee d) \wedge (y \vee d))$. Now there is the non-deterministic choice whether to first minimize $\exists a$ and then $\exists b$ or vice versa. Fig. 1 shows the quantifier trees for the two alternatives. Dependency schemes resulting from the trees (left and middle) are $D^l = \{(a, x), (x, c), (a, y), (b, y), (y, d)\}$ and $D^m = \{(b, x), (a, x), (x, c), (b, y), (y, d)\}$.

Apart from non-determinism, which has already been reported in [4,12,13,18], mini-scoping as well as quantifier scope analysis [13,18] is not optimal among syntactic methods for dependency computation. At this point, we informally introduce D^{std} and report its advantage over mini-scoping and scope analysis.

The standard dependency scheme D^{std}, which is the focus of our work, was defined in [25] and is based on ideas from expansion-based solvers [7,8]. Dependencies are identified by analyzing connections between variables in a PCNF over sequences of clauses as follows.

Definition 1 (X-path). *For $x, y \in V$, where V is the set of variables in the PCNF, and $X \subseteq V$, an X-path between x and y is a sequence $C_1, \ldots, C_k$ of clauses such that $x \in C_1$, $y \in C_k$ and $C_i \cap C_{i+1} \cap X \neq \emptyset$ for $1 \leq i < k$.*

Example 3. For the formula from Ex. 2, there are X-paths between b and y for $X = \{d\}$ and clauses $(b \vee d)$ and $(y \vee d)$, and between a and y for $X = \{b, d\}$ and clauses $(a \vee b)$, $(b \vee d)$ and $(y \vee d)$.

Definition 2 (D^{std} informally). *$(x, y) \in D^{\text{std}}$ whenever x and y are quantified differently and there is an X-path between x and y where X is the set of existential variables to the right of, but not adjacent to x in the quantifier prefix.*

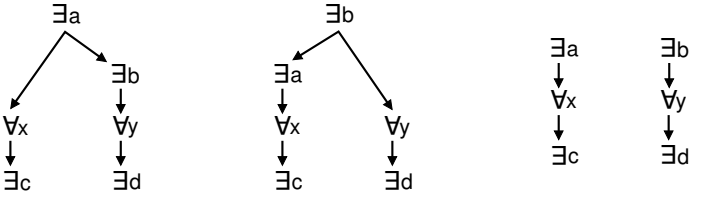


Fig. 1. Two possible quantifier trees for the QBF $\exists a, b \forall x, y \exists c, d.\ (a \vee x \vee c) \wedge (a \vee b) \wedge (b \vee d) \wedge (y \vee d)$ obtained by mini-scoping (left and middle) and dependencies by the standard dependency scheme D^{std} (right). See also Ex. 2 and 4.

A correctness proof of D^{std} is given in [25] and a formal definition in Def. 5.

Example 4. For the formula from Ex. 2, $D^{\text{std}} = \{(a,x),(x,c),(b,y),(y,d)\}$.

Note that in Ex. 4 $(a,y) \notin D^{\text{std}}$ and $(b,x) \notin D^{\text{std}}$, hence y does not depend on a and x not on b by D^{std}. By Def. 2, a and b are excluded from X, and there are no X-paths for $X = \{c,d\}$ between a, y and b, x in the QBF from Ex. 2.

Comparing dependencies from Ex. 2 and 4 shows a crucial difference between mini-scoping or scope analysis and D^{std}. Dependencies by D^{std} can be strictly less restrictive: no matter which of the two non-deterministically constructed quantifier trees (Fig. 1) are taken for dependency computation, either (a,y) or (b,x) is included in the resulting dependency set, but neither in D^{std}. The same applies to scope analysis like in [13,18] because any tree-shaped prefix of non-PCNF formulae can in principle be obtained by mini-scoping.

Because of non-determinism and more restrictive dependencies when using mini-scoping or scope analysis, we focus on D^{std}. Our motivation is two-fold. First, we want to extract a static graph representation of D^{std} from a QBF in PCNF. By traversing clauses in a QBF ϕ, $D^{\text{std}}(x)$ for *one* variable $x \in Var(\phi)$ can be computed in $O(|\phi|)$ time [25] where $|\phi|$ is the length of ϕ. However, computing $D^{\text{std}}(x)$ for *all* variables x by the same approach requires $O(|Var(\phi)|.|\phi|)$ time. We construct a directed acyclic graph (DAG) for D^{std}, which has the same worst-case time complexity but can be done efficiently in practice. The idea is similar to quantifier trees by mini-scoping [4] but does not suffer from non-determinism and, as shown, results in a less restrictive dependency relation.

Example 5. Search-based solvers profit from D^{std} because variables can be assigned earlier. In Fig. 1, both a and b have to be assigned before y (left tree) and before x (middle). By D^{std} (right), x and y can be assigned as soon as a, respectively b has been assigned.

Second, we aim at compactness in practice. We take advantage of properties of the connection relations from [7,8] which allow to merge existential variables into equivalence classes. A static connection relation over equivalence classes is defined which is shared between all variables, thus contributing to compactness.

In this work, we extend our ideas from [22] to existential variables, thus making our work applicable to arbitrary QBF solvers. We develop a formal background for a graph representation of D^{std} in Sec. 3 including proofs. Based on this theoretical part, practical aspects concerning dependency computation and graph construction are reported in Sec. 4. In Sec. 5, experimental results on structured formulae demonstrate the effectiveness of our approach.

2 Preliminaries

For a set of propositional variables V, a *literal* is either a variable $x \in V$ or its negation $\neg x$ where $v(x) = x$ and $v(\neg x) = x$ denotes the variable of a literal. A *clause* is a disjunction over literals. A propositional formula is in *conjunctive normal form (CNF)* if it consists of a conjunction over clauses.

A quantified boolean formula (QBF) $S_1 \ldots S_n . \phi$ in *prenex conjunctive normal form (PCNF)* consists of a propositional formula ϕ in CNF over a set of variables V and a *quantifier prefix* $S_1 \ldots S_n$. The quantifier prefix is a linearly ordered set of *scopes* S_i where $S_1 < \ldots < S_n$, which forms a partition on the set of variables: $V = S_1 \cup \ldots \cup S_n$ where $S_i \neq \emptyset$ and $S_i \cap S_j = \emptyset$ for $1 \leq i, j \leq n$ and $i \neq j$.

A scope S_i is *existential* if it is associated with an existential quantifier, written as $q(S_i) = \exists$ and *universal* otherwise where $q(S_i) = \forall$. The set of existential and universal variables is denoted by $V_\exists = \bigcup S_i$ for $q(S_i) = \exists$ and $V_\forall = \bigcup S_i$ for $q(S_i) = \forall$, respectively. For a variable $x \in S_i$, $s(x) = S_i$ is the scope of x and $q(x) = q(s(x))$ the *type* of x. For two adjacent scopes S_i and S_{i+1} where $1 \leq i < n$, $q(S_i) \neq q(S_{i+1})$. Given a QBF with n scopes, there are $n-1$ *quantifier alternations*.

For a scope S_i and literal l, $\delta(S_i) = i$ and $\delta(l) = \delta(s(v(l)))$ denote the *level* of S_i and of l, respectively. For scopes S_i, S_j and literals l, k, S_j is *larger* than S_i and k is larger than l if $\delta(S_i) < \delta(S_j)$ and $\delta(l) < \delta(k)$, respectively.

Let $R \subseteq V \times V$ be a binary relation on the set of variables V. The *reflexive and transitive closure* of R is the smallest reflexive and transitive $R' \subseteq V \times V$ such that $R \subseteq R'$. The *reflexive and transitive reduction* of R is the smallest $R' \subseteq V \times V$ such that R and R' have the same reflexive and transitive closure.

In the following, we consider QBFs in PCNF where for all clauses $C = (l_1 \vee \ldots \vee l_k)$, $v(l_i) \neq v(l_j)$ and $\delta(l_i) \leq \delta(l_j)$ for $1 \leq i < j \leq k$ and $q(v(l_k)) = \exists$. A clause neither contains multiple nor complementary literals of one and the same variable, all literals are sorted ascendingly according to their level and the largest literal is existential. *Universal reduction* [7,9] can be applied to remove literals l_k for which $q(v(l_k)) = \forall$. Furthermore, we assume that there occurs at least one literal for each $x \in V$ in the formula.

3 Theoretical Background

The goal of our work is a compact graph representation of the standard dependency scheme D^{std}. In this section we pick up our ideas from [22]. We first define a connection relation over equivalence classes of existential variables. A directed and reduced variant of this relation is tree-shaped and, as we prove, can be used for dependency computation by D^{std}. For reasons of space and conciseness we omit detailed proofs when appropriate. In definitions we explicitly state the types of variables since this is crucial particularly for connection relations.

Definition 3. *For $x \in V$, if $q(x) = \exists$ then $\overline{q(x)} = \forall$ and $\overline{q(x)} = \exists$ otherwise.*

Definition 4. *For a QBF and $q \in \{\exists, \forall\}$, $V_{q,i} = \{y \in V_q \mid \delta(y) \geq i\}$.*

Definition 5 (Standard Dependency Scheme). *For $x \in V, i = \delta(x) + 1$:* $D^{\text{std}}(x) = \{y \in V_{\overline{q(x)},i} \mid$ there is an X-path between x and y for $X = V_{\exists,i}\}$.

By setting $i = \delta(x) + 1$ and $X = V_{\exists,i}$, universal variables as well as variables from the scope of x are excluded from X as already informally in Def. 2.[1]

[1] The correctness proof of D^{std} in [25] is given for $i = \delta(x)$ and, according to the author's remarks, also works when $i = \delta(x) + 1$ as for our purposes.

i	$q(S_i)$	S_i	(a2, e5, e9)
1	∀	a1, a2	(e5, e9, e15)
2	∃	e3, e4, e5	(e3, e8, e13)
3	∀	a6, a7	(e4, a7, e10)
4	∃	e8, e9, e10	(e4, e13, e14)
5	∀	a11, a12	(a1, a6, e8, e14)
6	∃	e13, e14, e15	(a11, a12, e13)

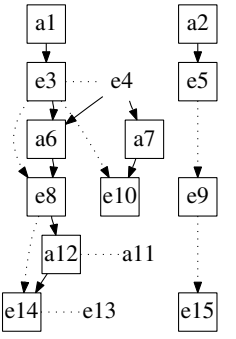


Fig. 2. QBF example. The table on the left shows the levels, quantifiers and variables for each scope in the first three columns and clauses as lists of literals in the last column. Variables and literals are uniquely identified by integers as in QDIMACS format [24]. Identifier prefixes "e" and "a" indicate types ∃ and ∀, respectively. The graph on the right shows a compact representation of D^{std} for the QBF (see also Ex. 10).

Example 6. For the QBF in Fig. 2, $\mathsf{e13} \in D^{\text{std}}(\mathsf{a1})$ by clauses $(\mathsf{a1}, \mathsf{a6}, \mathsf{e8}, \mathsf{e14})$ and $(\mathsf{e3}, \mathsf{e8}, \mathsf{e13})$, and $X = V_{\exists,2} = \{\mathsf{e3}, \mathsf{e4}, \mathsf{e5}, \mathsf{e8}, \mathsf{e9}, \mathsf{e10}, \mathsf{e13}, \mathsf{e14}, \mathsf{e15}\}$.

Different from [7,8,25], the following definition of connections is scope-aware.

Definition 6 (Connection). *For $x, y \in V$, x is* connected *to y with respect to scope S_i, written as $x \to_i y$, if, and only if $y \in V_{\exists,i}$ and there is a clause C such that $x \in C$ and $y \in C$. $\to_i^*$ denotes the reflexive and transitive closure of $\to_i$.*

Relation $\to_i^*$ is defined with respect to some scope S_i: if $x \to_i^* y$, then x is connected to y over existential variables from scopes larger than or equal to S_i only. There is a close correspondence between X-paths and $\to_i^*$.

Corollary 1. *For $x, y \in V$, if $x \to_i^* y$, then there is an X-path between x and y for $X = V_{\exists,i}$.*

Due to Def. 6 the converse of Cor. 1 does not hold in general. For example, if there is an X-path between $x \in V_\exists$ and $y \in V_\forall$ then $x \not\to_i^* y$ for all i. A weaker variant can be stated as follows.

Corollary 2. *For $x \in V, y \in V_\exists$, if there is an X-path between x and y for $X = V_{\exists,i}$ and $i \leq min(\delta(x), \delta(y))$, then $x \to_i^* y$.*

Connections with respect to a scope S_j are preserved for any smaller scope S_i.

Corollary 3. *For $x, y \in V, i \leq j$: if $x \to_j^* y$, then also $x \to_i^* y$.*

For proper values of i, connections between existential variables are symmetric because X-paths resulting from Cor. 1 can be reversed.

Lemma 1. *For $x, y \in V_\exists$ and $i \leq min(\delta(x), \delta(y))$: if $x \to_i^* y$ then $y \to_i^* x$.*

Example 7. For the QBF in Fig. 2, $\mathsf{e3} \to_4 \mathsf{e8}$ but $\mathsf{e3} \not\to_5 \mathsf{e8}$, $\mathsf{e8} \to_6 \mathsf{e14}$ and by Cor. 3 also $\mathsf{e8} \to_1 \mathsf{e14}$, further $\mathsf{e3} \to_2^* \mathsf{e14}$ and by Lem. 1 $\mathsf{e14} \to_2^* \mathsf{e3}$.

As a first step towards a compact representation of D^{std} we want to take advantage of situations where two variables can be regarded as equivalent.

Definition 7 (Equivalence). *For $x, y \in V$, x is* equivalent *to y, written as $x \approx y$, if, and only if either (1) $x = y$ or (2) $q(x) = q(y) = \exists$, $\delta(x) = \delta(y) = i$ and $x \rightarrow_i^* y$.*

Variables x and y are equivalent if $x = y$ or both are from the same existential scope S_i and are connected by existential variables larger than or equal to S_i.

Theorem 1. *$\approx$ is an equivalence relation. For $x \in V$, $[x]$ is the class of x.*

Proof. Reflexivity is trivial since $x \approx x$ for $x \in V$ by Def. 7. If *not* $q(x) = q(y) = \exists$ then by Def. 7 $x \approx y$ if, and only if $x = y$. Since $=$ is an equivalence relation, symmetry and transitivity of $\approx$ follow immediately. Otherwise, assume $q(x) = q(y) = \exists$. If $x \approx y$ and $x = y$, then also $y \approx x$ by Def. 7. If $x \approx y$ and $x \neq y$ then by Def. 6 and Def. 7 $\delta(x) = \delta(y)$ and $x \rightarrow_i^* y$ for $i = \delta(x) = \delta(y)$. Then by Lem. 1 also $y \rightarrow_i^* x$ and hence $y \approx x$. Therefore $\approx$ is symmetric. To show transitivity, assume $x \approx y'$ and $y' \approx y$ for $y' \in V$. Then more precisely $y' \in V_\exists$ (because otherwise $x \not\approx y'$ and $y' \not\approx y$) and by Def. 7 also $x \rightarrow_i^* y'$, $y' \rightarrow_i^* y$ for $i = \delta(x) = \delta(y') = \delta(y)$ and $q(x) = q(y') = q(y)$. By $x \rightarrow_i^* y'$, $y' \rightarrow_i^* y$ and transitivity of $\rightarrow_i^*$, also $x \rightarrow_i^* y$, hence $x \approx y$. □

Example 8. For the QBF in Fig. 2: $\mathsf{e3} \approx \mathsf{e4}$ since $q(\mathsf{e3}) = q(\mathsf{e4}) = \exists$, $\delta(\mathsf{e3}) = \delta(\mathsf{e4}) = 2$ and $\mathsf{e3} \rightarrow_2^* \mathsf{e4}$ by $\mathsf{e3} \rightarrow_2 \mathsf{e8} \rightarrow_2 \mathsf{e14} \rightarrow_2 \mathsf{e4}$. Also $\mathsf{e13} \approx \mathsf{e14}$ since $\mathsf{e13} \rightarrow_6 \mathsf{e14}$ but $\mathsf{e5} \not\approx \mathsf{e4}$ because $\mathsf{e5} \not\rightarrow_2^* \mathsf{e4}$. Trivially $\mathsf{a11} \approx \mathsf{a11}$ and $\mathsf{e3} \not\approx \mathsf{e14}$.

Relation $\rightarrow_i^*$ is compatible with $\approx$: if two variables are connected then so are all members of their respective classes and vice versa as stated in Lem. 2.

Lemma 2. *Let $x, y \in V, i \leq min(\delta(x), \delta(y))$. Then $x \rightarrow_i^* y$ if, and only if $x' \rightarrow_i^* y'$ for all $x' \in [x], y' \in [y]$.*

Proof. The proof works regardless of the types of x and y by Def. 6 (reflexivity of $\rightarrow_i^*$), Cor. 3 and Def. 7. Trivial cases arise for $V_\forall$. Assume $x \rightarrow_i^* y$ for $x, y \in V$ and $i \leq min(\delta(x), \delta(y))$. Then for $x' \in [x], y' \in [y]$, $x' \rightarrow_i^* x$ and $y \rightarrow_i^* y'$ by Cor. 3 and Def. 7. Since $x' \rightarrow_i^* x$, $x \rightarrow_i^* y$ (by assumption), $y \rightarrow_i^* y'$, also $x' \rightarrow_i^* y'$ by transitivity of $\rightarrow_i^*$. The other direction can be shown similarly by Lem. 1. □

When regarding $[x]$ as an arbitrary class member, we may write, for example, $[x] \rightarrow_i^* [y]$ by Lem. 2. This notation denotes connections between classes.

Lem. 2 would not hold for arbitrary values of i. For example, if $\delta(x) < i$ then $x \not\rightarrow_i^* x'$ for $x' \in [x]$, which contradicts Def. 7. The following variant of Lem. 2 does not refer to $[x]$ and holds for arbitrary values of i.

Lemma 3. *Let $x, y \in V$ with $\delta(x) \leq \delta(y)$. Then $x \rightarrow_i^* y$ if, and only if $x \rightarrow_i^* y'$ for all $y' \in [y]$.*

Example 9. For the QBF in Fig. 2, $\mathsf{e3} \approx \mathsf{e4}$, $\mathsf{e10} \approx \mathsf{e10}$, where $[\mathsf{e10}]$ is a singleton class, and $\mathsf{e4} \rightarrow_2^* \mathsf{e10}$ because $\mathsf{e4} \rightarrow_2 \mathsf{e10}$. By Lem. 2, also $\mathsf{e3} \rightarrow_2^* \mathsf{e10}$ because $\mathsf{e3} \rightarrow_2 \mathsf{e8} \rightarrow_2 \mathsf{e14} \rightarrow_2 \mathsf{e4} \rightarrow_2 \mathsf{e10}$.

Besides considering classes in $\rightarrow_i^*$ by Lem. 2, the following relation additionally allows to share information about connections, which is pointed out in Sec. 4.1.

Definition 8 (Directed Connection). $\rightsquigarrow^*$ *denotes the* directed connection relation. *For* $x \in V, y \in V_\exists$, $[x] \rightsquigarrow^* [y]$ *if, and only if,* $\delta(x) \leq \delta(y)$ *and* $x \rightarrow_i^* y$ *for* $i = \delta(x)$. *The reflexive and transitive reduction of* $\rightsquigarrow^*$ *is denoted by* $\rightsquigarrow$.

Corollary 4. *For* $x, y \in V$: *if* $[x] \rightsquigarrow^* [y]$ *then either* $[x] = [y]$ *or* $\delta(x) < \delta(y)$.

Relation $\rightsquigarrow^*$ is defined on classes only and respects the scope ordering. If $[x] \rightsquigarrow^* [y]$ then variables smaller than x are excluded in the connection between x and y. By Cor. 4, if $[x] \rightsquigarrow^* [y]$ then either x and y are in the same class or in different classes but from different scopes. We now prove that our definitions can be used to compute D^{std}.

Theorem 2 (Dependency Computation). *For* $x \in V, i = \delta(x) + 1$:

$$D^{\text{std}}(x) = \{y \in V_{\overline{q(x)},i} \mid \exists w \in V_{\exists,i} : x \rightarrow_i^* w \text{ and } y \rightarrow_i^* w\} \tag{1}$$

$$= \{y \in V_{\overline{q(x)},i} \mid \exists w \in V_{\exists,i} : x \rightarrow_i^* [w] \text{ and } [y] \rightarrow_i^* [w]\} \tag{2}$$

$$= \{y \in V_{\overline{q(x)},i} \mid \exists w \in V_{\exists,i} : x \rightarrow_i^* [w] \text{ and } [y] \rightsquigarrow^* [w]\} \tag{3}$$

Proof. Equivalence of left (LHS) and right-hand sides (RHS) of Eqn. 1 to 3.

- LHS(1) = RHS(1): Assume X-path P between x and y by clauses $C_1, \ldots, C_k$ where $y \in V_{\overline{q(x)},i}$. P can be split into P_1 between x, w for clauses $C_1, \ldots, C_j$ where $w \in C_j, 1 \leq j \leq k, w \in V_{\exists,i}$ and P_2 between w, y by clauses $C_j, \ldots, C_k$. By P_1 and Cor. 2 also $x \rightarrow_i^* w$ and by reversing P_2 and Cor. 2, also $y \rightarrow_i^* w$ and hence $y \in$ RHS(1). For the other direction, assume $x \rightarrow_i^* w$ and $y \rightarrow_i^* w$. Then by Cor. 1, there are X-paths P_1 between x, w and P_2 between y, w for $X = V_{\exists,i}$. An X-path P between x, y can be constructed by combining P_1 with reversed P_2, thus $y \in$ LHS(1).
- RHS(1) = RHS(2): Assume $x \rightarrow_i^* w$ and $y \rightarrow_i^* w$. Since $w \in V_{\exists,i}$, also $\delta(x) \leq \delta(w)$ and hence by Lem. 3 and Def. 7 also $x \rightarrow_i^* [w]$. Further, because $i \leq \delta(y)$ and $i \leq \delta(w)$ and hence $i \leq min(\delta(y), \delta(w))$, also $[y] \rightarrow_i^* [w]$ by Lem. 2 and Def. 7. Since $x \rightarrow_i^* [w]$ and $[y] \rightarrow_i^* [w]$, also $y \in$ RHS(2). For the other direction, assume $x \rightarrow_i^* [w]$ and $[y] \rightarrow_i^* [w]$. Similar arguments apply to derive $x \rightarrow_i^* w$ and $y \rightarrow_i^* w$ by Lem. 2, Lem. 3 and Def. 7. Hence $y \in$ RHS(1).
- RHS(2) = RHS(3): Assume $x \rightarrow_i^* [w]$ and $[y] \rightarrow_i^* [w]$. Since LHS(1) = RHS(1) = RHS(2), there is an X-path P between x, y for $X = V_{\exists,i}$ and clauses $C_1, \ldots, C_k$ where $y \in C_k$. Let l denote the largest literal in C_k. By assumptions in Sec. 2, $v(l) \in V_\exists$ and more precisely $\delta(y) \leq \delta(l)$ (if $q(y) = \forall$ then $\delta(y) < \delta(l)$). Assume that $w = v(l)$. Then $\delta(y) \leq \delta(w)$. By $y, w \in C_k$ also $y \rightarrow_j w$ for $j = \delta(y)$ and $y \rightarrow_j^* w$ by Def. 6. By $y \rightarrow_j^* w$ and $\delta(y) \leq \delta(w)$ also $[y] \rightsquigarrow^* [w]$. Since $x \rightarrow_i^* [w]$ and $[y] \rightsquigarrow^* [w]$ also $y \in$ RHS(3). For the other direction, Def. 8, Cor. 3 and Lem. 2 apply. □

4 Practical Application

In Thm. 2, Eqn. 1 is similar to computation by X-paths in Def. 5, Eqn. 2 refers to classes rather than individual variables, which is already an improvement. The step from Eqn. 2 to Eqn. 3 is the most interesting one for practical applications, yet this is not apparent from theory. Since $\rightsquigarrow^*$ is directed, it restricts the set of classes to be considered when connections of a variable are determined. In practice this contributes to compactness in addition to equivalence classes. In this section we first examine properties of $\rightsquigarrow^*$ over existential variables which allow to efficiently represent its reflexive and transitive reduction $\rightsquigarrow$ as a tree. This tree can be shared between all variables and is the basis for a graph data-structure representing D^{std}.

4.1 A Tree-Shaped Representation of $\rightsquigarrow$

Since $\rightsquigarrow^*$ is directed by Def. 8 and hence also antisymmetric and acyclic, its transitive reduction $\rightsquigarrow$ is unique [1]. The following lemma states a property of $\rightsquigarrow^*$ which accounts for the tree structure of $\rightsquigarrow$.

Lemma 4. *Let $x, y, z \in V_\exists$ where $\delta(x) \leq \delta(y)$. If $[x] \rightsquigarrow^* [z]$ and $[y] \rightsquigarrow^* [z]$ then $[x] \rightsquigarrow^* [y]$.*

Proof. Assume $[x] \rightsquigarrow^* [z]$ and $[y] \rightsquigarrow^* [z]$ where $\delta(x) \leq \delta(y)$. Then by Def. 8, $x \rightarrow_i^* z$ for $i = \delta(x)$ and $y \rightarrow_j^* z$ for $j = \delta(y)$ and $\delta(x) \leq \delta(y) \leq \delta(z)$. By Cor. 3 also $y \rightarrow_i^* z$ and by Lem. 1 $z \rightarrow_i^* y$. By Def. 6, $x \rightarrow_i^* z$ and $z \rightarrow_i^* y$, also $x \rightarrow_i^* y$ and $[x] \rightsquigarrow^* [y]$. □

If $[x] \rightsquigarrow^* [z]$ and $[y] \rightsquigarrow^* [z]$ for existential variables x, y, z and $\delta(x) \leq \delta(y)$ then by Lem. 4 $[x] \rightsquigarrow^* [z]$ is transitive. As a consequence $[x] \not\rightsquigarrow [z]$: at most one class is related to another one in $\rightsquigarrow$. Hence $\rightsquigarrow$ can directly be represented as a forest, that is a collection of trees.

Definition 9 (Connection Forest). *The* connection forest *(c-forest) for a QBF with m existential scopes is a collection of trees over $V_\exists$ with respect to $\approx$ with the following properties:*

1. *For $x, y \in V_\exists$: there is an edge $([x], [y])$ if, and only if $[x] \rightsquigarrow [y]$.*
2. *For $x, y \in V_\exists$: there is a path from $[x]$ to $[y]$ if, and only if $[x] \rightsquigarrow^* [y]$.*
3. *The maximum length (number of edges) of a path is $m - 1$ (by Cor. 4).*

4.2 Dependency Computation by Connection-Forests

The c-forest represents directed connections between existential variables. To compute $D^{\text{std}}(x)$ for arbitrary $x \in V$, a set of proper classes has to be found in the c-forest which exactly denote *all* connections of x to larger existential variables. Classes in such a set must be connected to x and be *minimal* with respect to the scope ordering since edges in the c-forest are directed. Descendants of such classes in the c-forest then comprise *all* connections of x by $\rightsquigarrow^*$.

Definition 10 (Smallest Ancestor). *For $y \in V_\exists, i \leq \delta(y)$ and the c-forest, let $h(i,[y]) = [y']$ such that $y' \in V_{\exists,i}, [y'] \rightsquigarrow^* [y]$ and there is no $y'' \in V_{\exists,i}$ with $i \leq \delta(y'') < \delta(y')$ and $[y''] \rightsquigarrow^* [y]$.*

Class $h(i,[y])$ is the smallest ancestor of $[y]$ which is larger than or equal to S_i, hence $h(i,[y])$ is minimal with respect to S_i and the scope ordering.

Definition 11 (Descendants). *For $x \in V$ and the c-forest, the set of descendants $H_i^*(x)$ with respect to scope S_i is defined as follows:*

1. $V_{C,i}(x) := \{[y] \mid y \in V_{\exists,i} \text{ and } x \rightarrow_i y\}$
2. $H_i(x) := \{[z] \mid [z] = h(i,[y]) \text{ for } [y] \in V_{C,i}(x)\}$
3. $H_i^*(x) := \{[y] \mid [z] \rightsquigarrow^* [y] \text{ for } [z] \in H_i(x)\}$

From clauses containing x, classes of existential variables larger than or equal to S_i are collected in $V_{C,i}(x)$. $H_i(x)$ contains smallest ancestors with respect to S_i for classes in $V_{C,i}(x)$. $H_i^*(x)$ comprises descendants of classes in $H_i(x)$ and represents all connections of x to existential variables larger than or equal to S_i.

Corollary 5. *For $x \in V$: if $[y] \in H_i^*(x)$ then $x \rightarrow_i^* y$.*

For $x \in V$, $H_i^*(x)$ exactly characterizes connections of x to existential variables. This is sufficient for computing $D^{\text{std}}(x)$. Informally, there is a dependence between two differently quantified variables if their sets of descendants in the c-forest are not disjoint.

Theorem 3 (Dependency Computation). *For $x \in V, i = \delta(x) + 1$:* $D^{\text{std}}(x) = \{y \in V_{\overline{q(x)},i} \mid H_i^*(x) \cap H_j^*(y) \neq \emptyset \text{ for } j = \delta(y)\}$.

Proof. Assume $x \in V$ and $i = \delta(x) + 1$. Direction $\supseteq$ follows right from Def. 11, Cor. 5, Cor. 3 and Thm. 2. To show $\subseteq$, assume $y \in D^{\text{std}}(x)$. Then there is an X-path P between x, y for $X = V_{\exists,i}$. Hence there are clauses $C_1, \ldots, C_k$ where $y, y_k \in C_k$ for some $y_k \in V_{\exists,i}$ with $\delta(y) \leq \delta(y_k)$. Such y_k always exists since by assumption the largest literal in a clause is existential.[2] Then P is also an X-path between x and y_k by $C_1, \ldots, C_k$ and hence $x \rightarrow_i^* y_k$ and $\delta(x) < \delta(y_k)$ since $i \leq \delta(y_k)$, $i = \delta(x) + 1$. We show that $[y_k] \in H_i^*(x) \cap H_j^*(y)$ for $j = \delta(y)$.

Since $y, y_k \in C_k$ by P, also $[y_k] \in V_{C,j}(y)$. Then $[z'] \in H_j(y)$ where $[z'] = h(j,[y_k])$ for $j = \delta(y)$. By Def. 10, $[z'] \rightsquigarrow^* [y_k]$, hence $[y_k] \in H_j^*(y)$.

Since P connects x and y_k, also $x, y_1 \in C_1$ for some $y_1 \in V_{\exists,i}$. Thus $[y_1] \in V_{C,i}(x)$ and $[z_1] \in H_i(x)$ for $[z_1] = h(i,[y_1])$. Then by Def. 10, $[z_1] \rightsquigarrow^* [y_1]$. P is also an X-path between y_1 and y_k by $C_1, \ldots, C_k$, hence $y_1 \rightarrow_i^* y_k$ and $\delta(x) < \delta(y_1), \delta(x) < \delta(y_k)$. Let w denote the smallest connecting variable in P between y_1, y_k: $m = \delta(w) = min(\{\delta(v) \mid v \in C_i \cap C_{i+1} \cap X, 1 \leq i < k\})$. Since m is minimal, also $y_1 \rightarrow_m^* w$, $w \rightarrow_m^* y_k$ and by Lem. 1 $w \rightarrow_m^* y_1$. By Def. 8 and since $m = \delta(w)$, also $[w] \rightsquigarrow^* [y_1]$, $[w] \rightsquigarrow^* [y_k]$. By Lem. 4, $[z_1] \rightsquigarrow^* [y_1]$ and $[w] \rightsquigarrow^* [y_1]$, also $[z_1] \rightsquigarrow^* [w]$. Then by $[z_1] \rightsquigarrow^* [w]$, $[w] \rightsquigarrow^* [y_k]$ and transitivity also $[z_1] \rightsquigarrow^* [y_k]$, hence $[y_k] \in H_i^*(x)$ because $[z_1] \in H_i(x)$. □

In contrast to Thm. 2, practical application follows right from Thm. 3. For a QBF, dependencies can be identified by checking descendants in the c-forest.

[2] If $x \in V_\forall$ then $y \in V_\exists$ and we may choose $y_k = y$.

4.3 A Graph Representation of D^{std}

We describe a static graph representation of D^{std} for a given QBF which is compact in practice. Representing each pair $(x, y) \in D^{\mathrm{std}}$ as a separate edge yields a graph with $|V|^2$ edges in the worst case. Instead, this can often be avoided by building the c-forest once and inserting edges as follows.

First, if $x \in V_\forall$ then by Thm. 3 any member y' of a class $[y] \in H_i^*(x)$ for $i = \delta(x) + 1$ depends on x (see also Thm. 3 in [22]). Thus the c-forest and set $H_i(x)$ compactly represent $D^{\mathrm{std}}(x)$ (see also Ex. 10). In the graph $H_i(x)$ is represented as edges from class $[x]$, which is singleton by Def. 7, to classes in the c-forest. After $H_i(x)$ for all $x \in V_\forall$ have been determined, universal classes $[y_1], [y_2]$ are merged whenever $H_j(y_1) = H_j(y_2)$ for $j = \delta(y_1)+1 = \delta(y_2)+1$ and either $H_j(y_1)$ or $H_j(y_2)$ is discarded. This reduces the number of edges in the graph. Such merging does *not* correspond to $\approx$ but is applied as post-processing.

Second, if $x \in V_\exists$ then edges for $y \in D^{\mathrm{std}}(x)$ need to be inserted explicitly in the graph. For $x \in V_\exists$ and descendant $[y'] \in H_i^*(x)$ where $i = \delta(x) + 1$, there is an edge from variable x to $[y]$ for $y \in V_{\forall,i}$ if $[y'] \in H_j(y)$ for $j = \delta(y) + 1$. This amounts to checking descendants in $H_i^*(x)$ and sets $H_j(y)$ for $y \in V_{\forall,i}$. Since universal classes have been merged before, again the number of inserted edges is reduced. Edges corresponding to transitive dependencies are discarded.

Example 10. In the graph in Fig. 2, boxes denote class representatives. Dotted vertical pointers like from [e3] to [e8] correspond to $\rightsquigarrow$ and denoted edges in the c-forest, dotted horizontal edges like between e13 and e14 connect class members and solid vertical pointers indicate dependencies. The classes of a11 and a12 have been merged in post-processing. The dependency $\mathsf{e15} \in D^{\mathrm{std}}(\mathsf{a2})$ is represented implicitly by the pointer from [a2] to [e5] and the path from [e5] to [e15]. Also $\mathsf{e13} \in D^{\mathrm{std}}(\mathsf{a11})$ by the pointer from [a12] to [e14] and $\mathsf{a11} \in D^{\mathrm{std}}(\mathsf{e8})$ by the pointer from [e8] to [a12]. Further $\mathsf{a7} \in D^{\mathrm{std}}(\mathsf{e4})$ by the pointer from e4 to [a7], but $\mathsf{a7} \notin D^{\mathrm{std}}(\mathsf{e3})$ since $[\mathsf{e10}] \notin H_3^*(\mathsf{e3})$.

5 Experimental Results

We have implemented a tool which constructs the graph representing D^{std} for a given QBF as described in Sec. 4.3. Tab. 1 shows experimental results with conclusions. In a first pass over the clauses, the c-forest is incrementally built by maintaining relation $\rightsquigarrow$ whenever pairs of existential literals l_1, l_2 are encountered in a clause. Additionally, sets $H_i(x)$ for $x \in V, i = \delta(x)+1$ are updated for literal pairs l_1, l_2 where $v(l_1) = x$, $\delta(l_1) < \delta(l_2)$ and either $q(v(l_1)) = q(v(l_2)) = \exists$ or $q(v(l_1)) = \forall$ and $q(v(l_2)) = \exists$. An efficient union-find data structure [27] is used to represent classes. In subsequent passes over the c-forest for $x \in V_\exists$, pointers representing dependencies of existential variables are inserted.

By using the c-forest over equivalence classes as basis for the graph of D^{std}, both time and memory requirements are kept small. For a given QBF ϕ, the graph can be constructed in $O(|V|.|\phi|)$ time and $O(|V|^2)$ space, when keeping edges for transitive dependencies. We observed that time required for removing

Table 1. Experimental results on publicly available, structured ("fixed" class) instances from QBF competitions 2005 to 2008 [15]. We did not include random instances. Experiments were run on 64-bit Ubuntu Linux 8.04, Intel® Q6700 at 2.66 GHz and 8 GB of memory. For reference, statistical data and a binary of our tool are available from `http://fmv.jku.at/qdag/`. For all formulae in the sets the graph for D^{std} has been built (see also Sec. 4.3 and 5 for comments on graph construction). The first line shows the numbers of formulae per set. Total run time, maximum over all formulae and average per formula are reported in seconds. Statistics are divided into two sections for existential and universal variables, respectively, and always $i = \delta(x)+1$. Maximum and average number of dependencies by D^{std} over all variables are shown. Compactness of the graph is indicated several times. For $x \in V_\forall$ classes in $H_i^*(x)$, which are reachable by ancestors in $H_i(x)$, efficiently represent $D^{\text{std}}(x)$. This becomes apparent when comparing $|H_i(x)|$, $|H_i^*(x)|$ and $|D^{\text{std}}(x)|$. For $x \in V_\exists$, $|H_i(x)|$ and $|H_i^*(x)|$ measure the effort for inserting dependency pointers since, starting from classes in $H_i(x)$, descendants in $H_i^*(x)$ are visited. Further, the average number of dependency classes per dependency for all $x \in V_\forall$ and $x \in V_\exists$, denoted by line $\frac{|\{[y]\in D^{\text{std}}(x)\}|}{|\{y\in D^{\text{std}}(x)\}|}$, is small. Note that classes result from $\approx$ for $x \in V_\exists$ and from post-processing for $x \in V_\forall$. The worst-case is 100%, where each dependency is in a singleton class. This is clearly not the case. The last line in each section shows the average number of classes per variable in each formula. Again, values are far below 100%, hence many variables can be regarded as equivalent.

	QBFEVAL'05	*QBFEVAL'06*	*QBFEVAL'07*	*QBFEVAL'08*
size	211	216	1136	3328
total time	7.94	1.35	227.05	300.31
max. time	0.58	0.03	7.96	8.11
avg. time	0.04	0.01	0.2	0.09
$x \in V_\forall$				
max. $\lvert D^{\text{std}}(x)\rvert$	256535	9993	2177280	2177280
avg. $\lvert D^{\text{std}}(x)\rvert$	82055.87	4794.60	33447.6	19807
max. $\lvert H_i(x)\rvert$	256	1	518	518
avg. $\lvert H_i(x)\rvert$	3.26	0.98	2.02	1.14
max. $\lvert H_i^*(x)\rvert$	797	5	797	1872
avg. $\lvert H_i^*(x)\rvert$	19.51	1.12	39.06	8.24
avg. $\frac{\lvert\{[y]\in D^{\text{std}}(x)\}\rvert}{\lvert\{y\in D^{\text{std}}(x)\}\rvert}$	3.44%	0.04%	6.42%	1.21%
classes per variables	28.2%	10.23%	40.31%	21.29%
$x \in V_\exists$				
max. $\lvert D^{\text{std}}(x)\rvert$	5040	440	5040	22696
avg. $\lvert D^{\text{std}}(x)\rvert$	12.76	2.98	3.24	4
max. $\lvert H_i(x)\rvert$	24	7	490	490
avg. $\lvert H_i(x)\rvert$	0.14	0.13	0.17	0.13
max. $\lvert H_i^*(x)\rvert$	797	7	797	1872
avg. $\lvert H_i^*(x)\rvert$	5.16	0.16	1.32	1.31
avg. $\frac{\lvert\{[y]\in D^{\text{std}}(x)\}\rvert}{\lvert\{y\in D^{\text{std}}(x)\}\rvert}$	2.37%	0.4%	2.76%	2.09%
classes per variables	10.96%	4.99%	11.45%	7.11%

transitive dependencies is negligible. As the results in Tab. 1 indicate, we achieve compaction of up to two orders of magnitude compared to a graph of D^{std} over variables rather than classes. This is due to the fact that connection information is shared between variables in the c-forest. To increase confidence in our implementation, we have run random tests and tests on formulae from Tab. 1 where we compared dependencies resulting from the graph to those from Def. 5.

6 Conclusion

Using less restrictive dependency schemes than those obtained from mini-scoping or scoping information readily available in structural formulae has the potential to boost performance of QBF solvers considerably. We gave a structural characterization of the simplest such formulation, based on the standard dependency scheme. The standard dependency scheme has so far only been applied in expansion based QBF solvers and preprocessing algorithms. As next step we want to incorporate our dependency analysis into search-based solvers, which currently are restricted to use tree-shaped prefixes. In a search-based solver it is prohibitive to recompute the dependency relation at each decision point. This also applies to static dependency representations based on mini-scoping such as quantifier trees [4]. As quantifier trees, our compact graph representation can be used as a precomputed approximation of actual dependencies. This can also be beneficial for expansion-based solvers.

Even though our algorithms can easily be extended to work on CNF with a tree-shaped prefix, it is not clear at this point how dependencies of variables introduced to encode structural QBF into CNF can be eliminated in order to lift our arguments to arbitrary structural QBF. This would also give us a way to experimentally show that less restrictive dependency schemes are useful for structural QBF solvers as well. As alternative one can try to generalize the concept of dependency schemes to structural formulas. Furthermore, we want to apply similar ideas to more advanced dependency schemes. Finally, we would like to thank Marko Samer for fruitful discussions on dependency schemes.

References

1. Aho, A.V., Garey, M.R., Ullman, J.D.: The Transitive Reduction of a Directed Graph. SIAM J. Comput. 1(2), 131–137 (1972)
2. Ayari, A., Basin, D.A.: QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In: Aagaard, M., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 187–201. Springer, Heidelberg (2002)
3. Bacchus, F., Walsh, T. (eds.): SAT 2005. LNCS, vol. 3569. Springer, Heidelberg (2005)
4. Benedetti, M.: Quantifier Trees for QBFs. In: [3], pp. 378–385
5. Benedetti, M.: sKizzo: A Suite to Evaluate and Certify QBFs. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 369–376. Springer, Heidelberg (2005)
6. Benedetti, M., Mangassarian, H.: QBF-Based Formal Verification: Experience and Perspectives. JSAT 5, 133–191 (2008)

7. Biere, A.: Resolve and Expand. In: Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 59–70. Springer, Heidelberg (2005)
8. Bubeck, U., Kleine Büning, H.: Bounded Universal Expansion for Preprocessing QBF. In: Marques-Silva, Sakallah (eds.) [23], pp. 244–257
9. Kleine Büning, H., Karpinski, M., Flögel, A.: Resolution for Quantified Boolean Formulas. Inf. Comput. 117(1), 12–18 (1995)
10. Cadoli, M., Giovanardi, A., Schaerf, M.: An Algorithm to Evaluate Quantified Boolean Formulae. In: AAAI/IAAI, pp. 262–267 (1998)
11. Dershowitz, N., Hanna, Z., Katz, J.: Bounded Model Checking with QBF. In: [3], pp. 408–414
12. Egly, U., Seidl, M., Tompits, H., Woltran, S., Zolda, M.: Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 214–228. Springer, Heidelberg (2004)
13. Egly, U., Seidl, M., Woltran, S.: A Solver for QBFs in Nonprenex Form. In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) ECAI. FAIA, vol. 141, pp. 477–481. IOS Press, Amsterdam (2006)
14. Giunchiglia, E., Narizzano, M., Tacchella, A.: QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 364–369. Springer, Heidelberg (2001)
15. Giunchiglia, E., Narizzano, M., Tacchella, A.: QBF Solver Evaluation Portal, 2001-2009, http://www.qbflib.org/index_eval.php
16. Giunchiglia, E., Narizzano, M., Tacchella, A.: Learning for Quantified Boolean Logic Satisfiability. In: AAAI/IAAI, pp. 649–654 (2002)
17. Giunchiglia, E., Narizzano, M., Tacchella, A.: Backjumping for Quantified Boolean Logic satisfiability. Artif. Intell. 145(1-2), 99–120 (2003)
18. Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantifier Structure in Search-Based Procedures for QBFs. TCAD 26(3), 497–507 (2007)
19. Jussila, T., Biere, A.: Compressing BMC Encodings with QBF. ENTCS 174(3), 45–56 (2007)
20. Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.M.: A First Step Towards a Unified Proof Checker for QBF. In: Marques-Silva, Sakallah (eds.) [23], pp. 201–214
21. Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In: Egly, U., Fermüller, C.G. (eds.) TABLEAUX 2002. LNCS, vol. 2381, pp. 160–175. Springer, Heidelberg (2002)
22. Lonsing, F., Biere, A.: Efficiently Representing Existential Dependency Sets for Expansion-based QBF Solvers. In: Proc. MEMICS, pp. 148–155 (2008)
23. Marques-Silva, J., Sakallah, K.A. (eds.): SAT 2007. LNCS, vol. 4501. Springer, Heidelberg (2007)
24. QBFLIB. QDIMACS Standard v1.1, http://www.qbflib.org/qdimacs.html
25. Samer, M., Szeider, S.: Backdoor Sets of Quantified Boolean Formulas. Journal of Automated Reasoning (JAR) 42(1), 77–97 (2009)
26. Stockmeyer, L.J., Meyer, A.R.: Word Problems Requiring Exponential Time: Preliminary Report. In: STOC, pp. 1–9. ACM, New York (1973)
27. Tarjan, R.E.: Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM 22(2), 215–225 (1975)
28. Zhang, L., Malik, S.: Conflict driven learning in a quantified Boolean Satisfiability solver. In: Pileggi, L.T., Kuehlmann, A. (eds.) ICCAD, pp. 442–449. ACM, New York (2002)

Beyond CNF: A Circuit-Based QBF Solver

Alexandra Goultiaeva, Vicki Iverson, and Fahiem Bacchus

Department of Computer Science
University of Toronto
{alexia,viverson,fbacchus}@cs.toronto.edu

Abstract. State-of-the-art solvers for Quantified Boolean Formulas (QBF) have employed many techniques from the field of Boolean Satisfiability (SAT) including the use of Conjunctive Normal Form (CNF) in representing the QBF formula. Although CNF has worked well for SAT solvers, recent work has pointed out some inherent problems with using CNF in QBF solvers.

In this paper, we describe a QBF solver, called CirQit (Cir-Q-it) that utilizes a circuit representation rather than CNF. The solver can exploit its circuit representation to avoid many of the problems of CNF. For example, we show how this approach generalizes some previously proposed techniques for overcoming the disadvantages of CNF for QBF solvers. We also show how important techniques like clause and cube learning can be made to work with a circuit representation. Finally, we empirically compare the resulting solver against other state-of-the-art QBF solvers, demonstrating that our approach can often outperform these solvers.

1 Introduction

QBF is a powerful generalization of SAT in which the variables can be universally or existentially quantified. While any problem in NP can be encoded in SAT, QBF allows us to encode any problem in PSPACE. This opens a much wider range of potential application areas for a QBF solver, including problems from areas like automated planning (particularly conformant and conditional planning), non-monotonic reasoning, electronic design automation, scheduling, model checking and verification, strategic decision making, and multi-agent scenarios, see for, e.g., [1,2,3].

State-of-the-art QBF solvers have utilized a number of techniques inherited from SAT solving technology. This has included the use of DPLL search augmented with clause learning along with additional QBF-specific techniques like solution backtracking and cube learning. Besides DPLL the original Davis-Putnam SAT solving technique [4] of ordered resolution has also been utilized [5], as well as methods involving the use of Skolemization to convert the QBF formula to SAT [6]. One constant in almost all of this work, however, has been the utilization of conjunctive normal form (CNF) in representing the QBF formula.

It has long been noted that conversion to CNF can lead to losing structure that could potentially be exploited computationally. As a result there has been some work on non-CNF SAT solvers, e.g., [7,8]. This work has shown that non-clausal representations can be effective for solving SAT. Nevertheless, the allure of CNF is that it can lead to very high performance implementations since it is a very simple and uniform representation.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 412–426, 2009.

Hence, the extra structure that can be exploited in a non-clausal representation has not been able to significantly outweigh the practical advantages of CNF in SAT solvers, and most SAT solvers continue to utilize CNF.

In QBF however the situation is different. In particular, for a similarly sized problem the search space explored by a QBF solver tends to be much larger than that explored by a SAT solver. Hence, there is much more potential for savings from exploiting the extra structure contained in non-clausal representations. In fact, there have been a number of papers that have identified various inadequacies of the CNF representation for QBF and proposed alternate representations aimed at addressing these problems, e.g., [9,10].

One of the most general and structure laden non-clausal representations is a circuit representation. Circuit representations have been used before in SAT solvers, e.g., [7,8], and in this paper we explore the use of this representation in a QBF solver.

One advantage of circuits is that they are more compatible with real problems—typically CNFs are generated from more structured representations like circuits. We also investigate ways of exploiting within our solver some of the extra structural information contained in the circuit. One particular example is the exploitation of don't care reasoning. We explain why don't care reasoning has more potential for efficiency gains when solving a QBF than when solving SAT. We also demonstrate how the essential techniques of unit propagation, clause learning, and cube learning used in CNF solvers can be adapted to a circuit representation. Finally, we explain how a circuit representation generalizes some of the key previously proposed techniques for addressing the inadequacies of CNF in the context of QBF.

We have implemented a solver we call **CirQit** (pronounced Cir-Q-it) that is based on our approach of utilizing a circuit representation. We are able to show empirically that it is very competitive with current state-of-the-art QBF solvers, and that on some problem suites it exhibits superior performance.

In the rest of the paper we first provide some essential background on QBF and the circuit representation of a QBF. We present some of the details of our circuit-based solver. Our solver utilizes a DPLL search procedure running on a circuit representation rather than on a CNF representation. We describe how propagation can be performed, and how clause and cube learning can be implemented. We discuss related work on QBF solvers based on non-clausal representations. Finally, we present various experimental results demonstrating the merit of our approach, and close with some conclusions.

2 Background

2.1 QBF

A QBF has the form $\boldsymbol{Q}.\phi$, where ϕ is an arbitrary propositional formula and $\boldsymbol{Q}$ is a sequence of quantified variables ($\forall x$ or $\exists x$). We require that the set of variables in ϕ be contained in $\boldsymbol{Q}$ so that $\boldsymbol{Q}.\phi$ has no free variables, and that ϕ contain only the connectives AND ($\wedge$), OR ($\vee$), and NOT ($\neg$).

A quantifier block qb of $\boldsymbol{Q}$ is a maximal contiguous subsequence of $\boldsymbol{Q}$ where every variable in qb has the same quantifier type. The quantifier blocks are ordered by their appearance in $\boldsymbol{Q}$: $qb_1 \leq qb_2$ iff qb_1 is equal to or appears before qb_2 in $\boldsymbol{Q}$. Each variable

x in ϕ appears in some quantifier block $qb(x)$. For two variables x and y we say that y is **downstream** of x (x is **upstream** of y) if $qb(y) > qb(x)$ ($qb(x) < qb(y)$). We also say that x is **universal** (**existential**) if its quantifier in $\boldsymbol{Q}$ is $\forall$ ($\exists$).

A QBF instance can be reduced by a literal ℓ (i.e., an assignment to one of its variables). The **reduction** of a formula $\boldsymbol{Q}.\phi$ by ℓ is denoted by $\boldsymbol{Q}.\phi|_\ell$. The reduction is the new formula $\boldsymbol{Q}.\phi'$ where ϕ' is ϕ with v replaced by the constant TRUE (if $\ell = v$) or FALSE (if $\ell = \neg v$), and optionally simplified according to standard logical rules: e.g., for any formula ψ, FALSE $\wedge\, \psi$ is equivalent to FALSE and $\forall x.\psi$ is equivalent to ψ if the variable x does not appear in ψ. A specific example is $\forall xz.\exists y.(\neg y \vee (x \wedge z)) \wedge \neg(x \vee z)|_{\neg x}$ which is equal to $\forall xz.\exists y.(\neg y \vee (\text{FALSE} \wedge z)) \wedge \neg(\text{FALSE} \vee z)$ which simplifies to $\forall z.\exists y.\neg y \wedge \neg z$.

Semantically, the truth or falsity of a QBF formula (with no free variables) can be defined recursively: (1) $\forall x\boldsymbol{Q}.\phi$ is true iff both $\boldsymbol{Q}.\phi|_x$ and $\boldsymbol{Q}.\phi|_{\neg x}$ are true, and (2) $\exists x\boldsymbol{Q}.\phi$ is true iff at least one of $\boldsymbol{Q}.\phi|_x$ or $\boldsymbol{Q}.\phi|_{\neg x}$ is true. By instantiating the quantified variables one by one, following the quantifier ordering, and substituting true or false into ϕ we arrive at either a QBF where ϕ simplifies to FALSE (which is a false QBF) or a QBF where ϕ simplifies to TRUE (which is a true QBF).

A circuit is a directed acyclic graph with a single sink where the nodes are logical gates and the edges are signal lines connecting the gates. Each gate is either an AND, OR, or NOT gate, has a single outgoing output line, and one or more incoming input lines. The output line of the sink gate is the circuit output, and the lines that are not outputs of any gate are the circuit inputs. A circuit representation $\boldsymbol{Q}.C$ for the QBF formula $\boldsymbol{Q}.\phi$ is a circuit C where the variables in $\boldsymbol{Q}$ are in 1-1 correspondence with the circuit inputs. C can be constructed recursively as follows. If ϕ is a variable x, then C has only one line labeled by the variable x and no gates. If $\phi = \neg\psi$, then C consists of the circuit representing ψ with the output of this circuit connected to the input of a NOT gate. If $\phi = \psi_1 \wedge \cdots \wedge \psi_i$ ($\psi_1 \vee \cdots \vee \psi_i$), then C consists of the outputs of the circuits representing ψ_1 to ψ_i connected as inputs of an AND (OR) gate. One key feature of the circuit representation is that duplicated sub-formulas in ϕ can be represented by a single subcircuit—the output line of that subcircuit can be used as an input in all places the sub-formula appears.

The lines of a circuit can take on the values TRUE or FALSE, and these values can be propagated to other lines of the circuit using standard rules of Boolean logic. For example, if an input line of an AND gate has value FALSE then FALSE can be propagated to the output line of the gate. A circuit $\boldsymbol{Q}.C$ **represents a formula** $\boldsymbol{Q}.\phi$ when for any setting of the variables in ϕ, ϕ will simplify to TRUE (FALSE) if and only if TRUE (FALSE) is propagated to the output of C given the same setting for its corresponding input lines. The construction described above yields a circuit C that represents ϕ.

Hence, we can evaluate a QBF formula $\boldsymbol{Q}.\phi$ by constructing a circuit $\boldsymbol{Q}.C$ representing it, and then evaluating the previously given definition of truth for a QBF formula by propagating values in C. That is, we can detect when ϕ simplifies to TRUE or FALSE by detecting when TRUE or FALSE is propagated to the output line of C.

3 A Circuit-Based Solver

Similar to previous circuit based SAT solvers, e.g., [7,8], our solver utilizes DPLL search to determine the truth or falsity of the QBF $\boldsymbol{Q}.\phi$. Specifically, ϕ is represented as a circuit C with the variables in $\boldsymbol{Q}$ being the inputs to C. During DPLL search, these variables are branched on in an order respecting the quantifier ordering (i.e., if x is upstream of y then the search must branch on x before y). Each branch sets a variable of ϕ and hence a corresponding input line of C. The input line values are propagated through C, and the search verifies that at least one side each existential branch and both sides of each universal branch lead to a true circuit output (i.e., satisfies ϕ).

However, to make this process more efficient, e.g., to detect when certain input lines must take on a particular value for the circuit output to be TRUE, the solver initially sets the circuit output line to TRUE and propagates values backwards in the circuit as well as forwards. Backwards propagation, like forward propagation, follows the rules of Boolean logic, e.g., if an OR gate's output line is set to FALSE then FALSE can be propagated to all of its input lines. With backward propagation from a TRUE output we can detect when ϕ is falsified (i.e., when a setting of the input lines would lead to FALSE being propagated to the output line) by the occurrence of a conflict where both TRUE and FALSE is propagated to some line in the circuit. Such conflicts also allow us to employ 1UIP clause learning techniques on the circuit representation.

One negative aspect of fixing the circuit output line to TRUE, however, is that we can no longer use the propagation of TRUE to the circuit output to detect when the formula ϕ has become satisfied by the current setting of its variables—the output line always has that value. We discuss below how this problem is resolved in our solver by utilizing information gathered during don't care propagation.

3.1 Propagation

Our implementation of forward and backward propagation in the circuit is based on a previous circuit based SAT solver described in [7]. In that paper Thiffault et al. showed that this kind of propagation in the circuit corresponds in a precise way to Unit Propagation (UP) on an equivalent CNF encoding of the formula. In particular, if the circuit was converted to CNF using the standard Tseitin encoding [11], then corresponding to each circuit line l there would be a new variable v in the CNF encoding such that a value would be propagated to the line l in the circuit if and only if UP in the CNF forces v to take the same value. Besides this basic mechanism, however, our QBF solver differs from previous circuit based SAT solvers in a number of ways.

Representing internal lines. The CNF encoding of a formula introduces additional variables that correspond to the sub-formulas of the formula. These additional variables are very useful in a SAT solver as they can be branched on in a DPLL search (implicitly positing a truth value for an entire sub-formula), and they can be included in learnt clauses increasing the effectiveness of clause learning.

A key feature of our circuit based QBF solver is that it also utilizes the learning techniques common in DPLL based QBF solvers that employ CNF [12] i.e., clause and cube learning. Hence, to facilitate the power of the learnt clauses we introduce additional variables to label the internal lines of the circuit, as is done in circuit based SAT solvers. (The

input lines are all labeled with a variable of the original formula $\boldsymbol{Q}.\phi$). Formally, all of these new variables are existential, and we place them as early in the quantifier ordering as possible. Specifically, each internal line l in the circuit is the output line of a sub-circuit c that has some set of input lines representing a set of variables V of ϕ. We place the new variable representing l in the quantifier prefix immediately after the last variable of V in the prefix. By placing these new variables as early as possible in the quantifier prefix we enable more effective universal reduction during clause learning and propagation.

Note however that in QBF, unlike SAT, these new variables are never branched on during the DPLL search. The DPLL search must respect the quantification order when selecting variables to branch on, so by the time it can select a variable v representing the output of sub-circuit c all of the inputs to c must have already been assigned, and hence v would already be assigned by propagation.

Universal Reduction. In CNF represented QBF formulas universal reduction is a powerful additional rule of inference that enables further unit propagation and conflict detection. We say that a universal variable is **tailing** in a clause if it is downstream of all existentials in the clause. Universal reduction is the rule of inference where all tailing universals can be removed from a clause. It can be applied during search: when an existential in a clause is falsified and thus removed from the clause some universal in the clause might become tailing and thus removable by universal reduction.

There are two cases where universal reduction can reduce DPLL search. First, it can be used to infer a conflict when a clause contains only universal variables: by universal reduction we can reduce any such clause to the empty clause. Second, it can be used to infer unit clauses when a clause becomes unit after universal reduction. In this case it must be that the clause contains a single existential variable e with all other variables in the clause being universal and downstream of e.

Our solver can detect the same set of conflicts and unit propagants arising from universal reduction as would be detected in CNF representation. Two additional propagation rules are utilized to achieve this. The first rule is triggered whenever there is a gate g such that (a) the output line of g has been assigned some value TF, (b) TF is not entailed by g's assigned input lines (e.g., if g is an OR gate, TF $=$ TRUE, and none of g's assigned inputs are TRUE), and (c) all of g's unassigned input lines are universally quantified. In this case we have a conflict corresponding to the generation of a clause containing only universals. The second rule is triggered whenever there is a gate g such that conditions (a) and (b) as above hold, and (c) g's unassigned input lines contain only a single existential line e and all of the other unassigned lines, which are hence universal, are downstream of e. In this case we can force e to take on a value that entails TF. This corresponds to the generation of a unit clause after universal reduction. For example, if g is an AND gate, TF $=$ FALSE, and all of g's other assigned inputs are TRUE, then e is forced to be FALSE.

3.2 Don't Care Propagation

An important way in which the circuit structure can be exploited is via don't care reasoning. For example, when one input of an OR gate is set to TRUE, the other inputs become irrelevant to its output value. By detecting the variables that have become irrelevant to all the gates they feed into, DPLL can avoid branching on them. Don't care

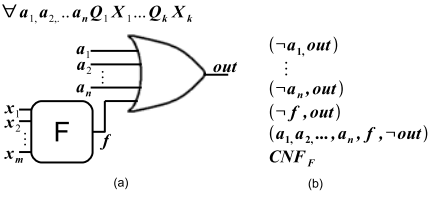


Fig. 1. Circuit and Equivalent CNF Encoding

propagation detects such variables and we implement don't care propagation in our solver using the techniques developed in [7].

Don't care propagation can be useful in SAT, but it has even more potential to be helpful in QBF due to repetitions caused by universal variables. To illustrate, consider the circuit in Figure 1, where $Q_1X_1...Q_kX_k$ represents an arbitrary set of quantifiers over the variables $x_1, ..., x_m$, and F is an arbitrarily complicated boolean circuit. It can be seen that any variable assignment with at least one of $a_1, ..., a_n$ set to TRUE makes the circuit output TRUE. In our solver, as soon as one of the variables a_i is set to TRUE, all x_i variables can be recognized as irrelevant, so there is only one setting of $a_1, ...a_n$ for which the solver actually branches on any x_i variables.

A CNF based solver, on the other hand, would have the CNF representation shown in Figure 1b, where CNF_F represents the clausal encoding of F. If any of the a_i variables are set TRUE, then out is also set to TRUE, and all clauses disappear except for those in CNF_F. The solver will then have to continue branching on the x_i variables until a solution is found that satisfies all clauses in CNF_F, a potentially difficult task. Furthermore, we see that the solver can unnecessarily try to satisfy the clauses in CNF_F $2^n - 1$ times. A solver exploiting learning might solve these repetitions more efficiently, but can still perform many unnecessary branching operations. It is this repetition from universal variables that makes don't care reasoning more effective in QBF.

Don't care propagation is achieved by detecting when gate outputs are **justified**. A gate in the circuit is justified when its assigned inputs are sufficient to imply its output. Once a gate is justified, its unassigned inputs have no effect, so they become *irrelevant* with respect to that gate. If a line becomes irrelevant with respect to all of the gates it is an input of, it becomes a **don't care**, meaning its value has no effect on the circuit output. Further, if the don't care line is a gate output all of its unset inputs can in turn be marked as irrelevant with respect to it, which might generate another round of don't care propagation. Since these don't care variables have no effect on the circuit the DPLL search engine need never branch on them. For example, in Figure 1 once one of the a_i inputs is set to TRUE, all remaining unassigned inputs to the final OR gate will be marked as don't care: they have all become irrelevant with respect to that gate and this is the only gate they are an input to. Don't cares can then be propagated back through all of the sub-circuit F until all of the x_i are marked as don't care. After this, DPLL can detect that it need not branch on any other variables as all remaining unassigned variables (input lines and internal lines) have become don't care.

3.3 Clause Learning

Following [7] we implement clause learning in our solver by computing a clausal reason from the circuit structure for every line that is assigned by propagation. As DPLL branches on variables that correspond to input lines of the circuit, propagation is used to set other lines in the circuit. Since each circuit line is represented by an existential variable, propagating a value to these lines corresponds to forcing a literal representing the assignment of this value to the line's corresponding variable. The logical structure that allowed the value to be propagated can then be used to construct a clausal reason for that forced literal. For example, if g is an AND gate with its output o set to FALSE, all of its assigned inputs $a_1, \ldots, a_k$, set to TRUE, and with unassigned inputs $e, u_1, \ldots, u_m$ where e is existential, the u_i are universal, and e is upstream of all of the u_i, then propagation will set e to FALSE. In this case we can extract from the circuit the clause $(\neg e, \neg u_1, \ldots, \neg u_m, \neg a_1, \ldots, \neg a_k, o)$ as the clausal reason for $\neg e$. Hence, on the trail of the DPLL search engine every forced literal can be given an associated clausal reason. Note that these clausal reasons are like the clauses that a QBF solver using a CNF representation would use to label its unit propagated literals.

In a similar way when conflicts are detected in the circuit a conflict clause can be constructed and returned to the DPLL search engine. For example, if g is an OR gate with its output o set to TRUE, with assigned inputs $a_1, \ldots, a_k$ all set to FALSE, and unassigned inputs $u_1, \ldots, u_m$ all of which are universal, then a conflict corresponding to the detection of an all universal clause is detected. From this conflict the conflict clause $(\neg o, a_1, \ldots, a_k, u_1, \ldots, u_m)$ can be constructed and returned to the DPLL search engine. With a CNF representation this is the clause that the current assignments would have reduced to an all universal clause. The case where a line has both TRUE and FALSE propagated to it can be handled in a similar fashion.

With conflict clauses to seed the process, and all forced literals on the trail labeled with clausal reasons, our solver can proceed to perform 1-UIP clause learning in the manner standard to DPLL-based QBF solvers and to use these clauses to non-chronologically backtrack the DPLL search. Finally, the solver can employ unit propagation over the learnt clauses in conjunction with propagation in the circuit, using literals forced by unit propagation to set lines and do further propagation in the circuit, and using lines set in the circuit to initiate further unit propagation in the learnt clauses.

3.4 Cube Learning

As mentioned above, because the circuit output O is initially set to TRUE we cannot use the propagation of TRUE to O to detect that the formula has become satisfied by the current set of variable assignments. Nevertheless, we can employ don't care propagation to detect circuit (formula) satisfaction. In particular, when all variables that are not marked as being don't care have been assigned and no conflicts have been generated, we know that the circuit is satisfied by the current set of assignments. Say we had not initially set the circuit output to TRUE. It can then be observed that whenever the assigned circuit input lines suffice to propagate TRUE to the circuit output, all remaining unset lines in the circuit (both internal and input lines) become don't care. It can be further observed that the don't care propagation mechanism outlined above will successfully label these unset lines as don't care.

Once the formula has been satisfied by the current variable assignments, we would like to perform cube learning. This involves finding a **subset** of the current variable assignments that are sufficient to satisfy the formula. Such a subset forms a base cube that can then be stored in a cube database, triggered in other parts of the DPLL search, and resolved with other cubes during search to generate more powerful cubes. A key element in making cube learning effective is to be able to generate small base cubes.[1]

With CNF representations base cubes must contain at least one true literal from each clause in the theory. With a circuit representation, however, finding a subset of variable assignments sufficient to satisfy the formula corresponds to finding a subset of the circuit inputs whose assigned values suffice to propagate TRUE to the circuit output. Don't care propagation helps in constructing small base cubes, as it eliminates from consideration all circuit inputs marked as don't care.

The algorithm we use in our solver is specified in Algorithm 1. The algorithm is initially called with the circuit's output as its input argument, and it involves sweeping through the circuit from the output to inputs picking a set of lines whose assigned values suffices to support the circuit output. Starting with the output gate, the algorithm selects a set of input lines that support the gate output. Then it continues on to find supports for the selected input lines. For example, if the gate is an AND gate with output set to FALSE then only one false input line is needed as support.

We note that we need not consider any don't care lines, all of the gate output lines the algorithm encounters have to be justified. To guide the selection of a supporting input, for each gate output line we memorize the input line that was responsible for it first becoming justified. For gate output line l we use $l.justfiedReason$ to denote this input line.

This approach for selecting a supporting input for each gate output has two advantages. First, it is very efficient to implement: the cube can be recovered in a single pass of the circuit. Second, it favours adding the earliest-set variables to the base cube which sometimes allows the solver to backtrack further.

In the algorithm specification we also use $l.gateType$ to denote the gate type that l is an output for. If l is an input line (and hence not associated with a gate) we let $l.gateType$ be equal to INPUT. Finally, let $l.inputs$ denote input lines of the gate that l is an output, and let $l.val$ denote the value assigned to l.

4 Related Work

4.1 CCDNF

In [10] Zhang proposed adding to the CNF encoding of the QBF a redundant DNF encoding, creating a Combined Conjunctive and Disjunctive Normal Form (CCDNF). The aim of the DNF encoding was to overcome the inability of CNF to easily detect when the formula becomes satisfied. The DNF allowed the resulting solver to detect solutions earlier, without needing to assign all variables in the formula.

In some cases, our circuit based solver achieves similar early solution detection through its don't care propagation. In particular, once a partial assignment is sufficient to imply the circuit output, all remaining variables will be marked as don't care and

[1] Other heuristic considerations come into play, but space precludes discussing them here.

Algorithm 1: RecoverCube—Construct a Base Cube from a Circuit

```
1  RecoverCube (l)
   // Return a set of supporting input lines
2  begin
3      if l.gateType = INPUT then
4          return {l}
5      else if l.gateType = NOT then
6          return RecoverCube (l.inputs)
       else if ( (l.gateType = AND and l.val = TRUE)
7      or (l.gateType = OR and l.val = FALSE) ) then
8          S = ∅ // Consider all children
9          foreach c ∈ l.inputs do
10             S = S∪ RecoverCube (c)
11         end
12         return S
13     else
           // root is a False AND gate or a True OR gate
           // Can select one child that is assigned the same
              value
14         return RecoverCube (l.justfiedReason)
15 end
```

the search engine can immediately backtrack. However, when a solution is detected, our solver must execute Algorithm 1 to extract a base cube—with the DNF encoding this computation is not needed, the information contained in the base cube is already encoded in the triggered DNF. Also the DNF encoding contains auxiliary variables that can make the cubes more compact, and perhaps more powerful. In our solver all base cubes contain input variables only.

However, the circuit representation has some advantages over IQTest. It preserves more problem structure than the CCDNF encoding. Potentially, additional ways can be discovered for further exploiting this structure. Also don't care propagation allows us to avoid branching on irrelevant variables. IQTest, on the other hand, has no way of determining when a variable is irrelevant, and can still branch on such variables prior to finding a solution. Thus, our solver can sometimes make fewer decisions during search.

Also, while making its conversion, IQTest creates two different sets of auxiliary variables: one set for the CNF, and another one for DNF representations. This limits the amount of knowledge sharing between the two representations. The circuit representation has only one set of auxiliary variables (variables representing the internal lines), so that the different modes of reasoning share the same representation.

Nevertheless, given that the circuit representation contains all of the information used to generate the DNF encoding, it is possible that the computational advantages of the DNF encoding can be captured directly from the circuit representation. We plan to investigate this possibility in future work.

The empirical results in the next section show that while IQTest outperforms our solver on some benchmarks, there are domains where the advantages of the circuit representation are evident.

4.2 Don't Care Literals

In [13], the authors augment the CNF encoding by adding *don't care literals* to clauses so they can be marked as redundant when the don't care literals become true. This is effectively the same as our solver marking a circuit line as redundant, but as they point out, they are unable to encode all don't care conditions. By dynamically detecting don't care conditions as they occur, we are able to detect more don't care variables during search than their static method.

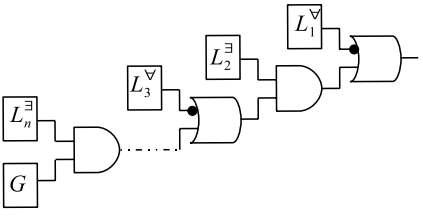


Fig. 2. Adversarial game encoding

4.3 Dual CNF and DNF

[9] created a dual CNF-DNF encoding geared towards addressing the inadequacies of CNF when encoding adversarial games. Their approach is to encode the rules for the universal player in a DNF. Then, if the universal player ever violates the rules, the DNF portion is detected to be true, and the existential player is declared the winner.

The main benefit of their approach—determining when the universal player cheats—is easily achieved in our solver by exploiting a circuit representation. Figure 2 shows an example of a circuit encoding an arbitrary two player game with n turns [14]. A box labeled $L_i^{\exists}$ represents a sub-circuit encoding the rules for the existential player in move i, while $L_j^{\forall}$ encodes the rules for the universal player in move j. At any move, if the universal player violates their rules, all remaining moves in the game become don't care, and the existential player is declared the winner (i.e., TRUE is propagated to the circuit output).

4.4 Negation Normal Form

In [15] the authors discuss a solver qpro for formulas in Negation Normal Form (NNF). The main focus of qpro is relaxing the restriction of a prenex form. This is orthogonal to our approach, and the circuit solver can be extended in a similar manner.

However, even without explicitly dealing with non-prenex formulas, don't care propagation together with clause and cube learning can often achieve similar results, and our solver is quite competitive with qpro even in the domains with very short but wide quantifier trees. This is demonstrated in the experimental results.

The backtracking technique of qpro—relevance sets—involves the solver computing the set of variables whose values determined the truth or falsity of the formula, and allows the solver to backtrack non-chronologically over irrelevant variables. The idea of identifying relevant sets underlies the notions of clause and cube learning. Learnt

cubes and clauses also allow a solver to backtrack non-chronologically over unrelated variables, with the added benefit that the learnt cubes and clauses can be utilized in the rest of the search. Since our solver implements cube and clause learning it does not need to compute relevance sets.

5 Experimental Results

Our solver CirQit implements the ideas described in this paper. Its input is a circuit description in ISCAS-85 format using AND, OR and NOT gates, along with the quantifier prefix. The solver first simplifies the circuit by merging identical subformulas. It then solves the circuit using DPLL search running on the circuit representation as described above.

Table 1. Comparison between CirQit and other state-of-the-art non-CNF non-Prenex solvers. The largest number of instances solved is shown in **bold**, with ties broken by the time taken to solve those instances.

Benchmark Families	CirQit		qpro		pQBF	
(number of instances)	Solved	Time	Solved	Time	Solved	Time
Seidl (150)	147	2,281	**150**	7	13	3,326
assertion (120)	**3**	1	1	0	0	0
consistency (10)	0	0	0	0	0	0
counter (45)	**39**	1,315	31	126	31	161
dme (11)	**10**	15	10	1,193	5	287
possibility (120)	**10**	1,707	0	0	0	0
ring (20)	**15**	60	9	397	9	158
semaphore (16)	**16**	7	16	91	16	726
Total (492)	**240**	5,389	217	1,816	74	4,660

We compared CirQit with state-of-the art CNF and non-CNF solvers on all the non-Prenex, non-CNF benchmarks currently available from QBFLIB [16]. Unless otherwise stated, all tests were run on a 2.8GHz machine with 12GB of RAM. The results display the number of problems that each solver was able to solve within the time limit of 1200 CPU seconds per instance, and the total time taken for all the **solved** instances, rounded down to the nearest second.

Table 1 shows the comparison against the two top solvers from the non-prenex non-CNF track of the QBFEVAL'08 competition. One of the solvers is qpro (discussed above), version of 29.02.08 available from the authors' site. The other one is pQBF [17].[2] The benchmarks, originally in QBF1.0 format, were converted into ISCAS-85 format for CirQit and into *pro* format for qpro. Conversion time was negligible and was not included in the results.

[2] On some instances pQBF gave a *parser stack overflow* error. In a few cases, it proceeded to return an answer. This happened on large instances in benchmark families for which pQBF timed out on smaller problems. The answer returned under these circumstances was always FALSE, and on at least one instance it was confirmed to be incorrect by multiple other solvers. This led us to believe that this answer was returned in error. The results presented here consider such instances as failure cases for pQBF.

Table 2. Comparison between CirQit and other state-of-the-art CNF-based solvers. The largest number of instances solved is shown in **bold**, with ties broken by the time taken to solve those instances.

Benchmark Families	CirQit		sKizzo		2clsQ		yquaffle		quantor		Qube	
(number of instances)	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time
Seidl (150)	**147**	2281	37	6,301	0	0	0	0	42	3,272	144	4,688
assertion (120)	3	1	14	796	49	7,035	23	114	**119**	8,736	3	0
consistency (10)	0	0	1	40	0	0	0	0	**10**	720	0	0
counter (45)	**39**	1,315	34	1,185	30	89	31	1,077	28	414	29	1225
dme (11)	**10**	15	0	0	0	0	0	0	0	0	6	75
possibility (120)	10	1,707	13	700	13	1,666	10	505	**111**	7,976	10	25
ring (20)	**15**	60	12	752	11	1,048	12	607	11	479	15	1,781
semaphore (16)	**16**	7	14	68	13	47	7	261	16	12	14	1,833
Total (492)	240	5,389	125	9,844	116	9,888	83	2,566	**337**	21,613	212	9,629

qpro outperforms CirQit on only the Seidl dataset (shown in Table 1). This dataset contains problem instances that typically have short but wide quantifier trees, a structure that qpro is particularly well suited to exploit. Although CirQit performs worse than qpro on this dataset, we can see that it outperforms all the other solvers (including the CNF-based solvers discussed below), and is one of only two solvers that come close to qpro on this dataset.

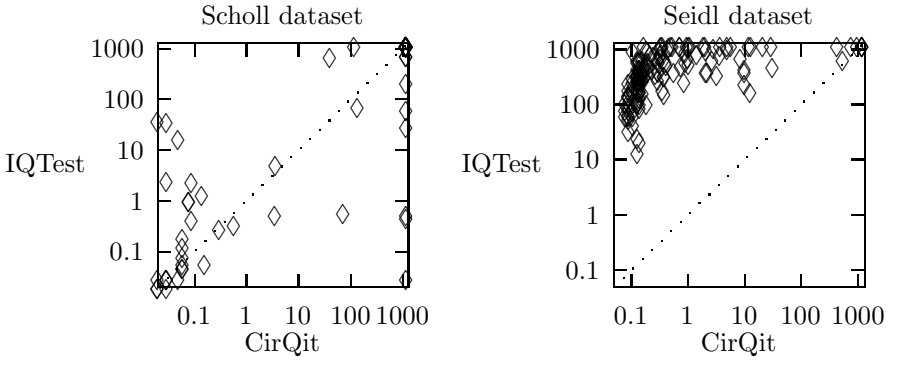
Other than the Seidl dataset, CirQit dominates the other two non-CNF solvers: it was able to solve all the problems that they solved, and also some additional ones.

Table 2 compares CirQit against a number of CNF-based solvers. In order to apply the CNF-based solvers, the benchmarks were converted from QBF1.0 to qdimacs format using the translator available from QBFLIB webpage. Again, the conversion times were not included.

The solvers tested were sKizzo (v0.8.2) [18], 2clsQ [19], yQuaffle (version 21006) [12], quantor (version 3.0, with the recommended picosat back end) [5] and Qube (version 6.1) [20]. These solvers are state-of-the-art QBF solvers as shown by QBFEVAL competition results. The predecessors of solvers sKizzo and Quantor were the best two solvers at QBFEVAL'05; 2clsQ and sKizzo took first and second places at QBFEVAL'06; Qube won QBFEVAL'07 with yQuaffle being the next-best standalone solver (disregarding the solvers based on a portfolio approach), and Qube6.1 was the best standalone solver at the QBFEVAL'08 competition– second only to a solver based on a portfolio approach.

Comparing with the CNF solvers, we see that CirQit is quite competitive with them. It outperforms all of the CNF solvers on a number of domains. For domains counter and dme, CirQit is able to solve a number of problems that no CNF-based solver could solve; for each of ring and semaphore domains, CirQit is tied with one CNF-based solver (Qube and quantor, respectively) on the number of solved instances but wins based on the time taken to solve them, and does notably better than the other solvers.

The domains on which CirQit does not outperform the CNF solvers are the assertion, consistency and possibility domains, which are all part of the set *BMC_QBF_1.0*. Note that all of the non-CNF solvers perform badly on this benchmark set. On these problems, Quantor is the clear winner. It also far outperforms all the other solvers on these benchmarks. Quantor does not employ DPLL search, using instead a combination



(a) Scholl dataset. (b) Seidl dataset

Fig. 3. Time comparison between CirQit and IQTest on two benchmark sets

of resolution and universal quantification to reduce the formula. Clearly this approach is better for these problems than any form of DPLL search.

Finally, we compared our solver with the previously discussed IQTest [10], which uses both a CNF and an DNF encoding of the problem. We were not able to perform a comprehensive comparison with IQTest due to the fact that IQTest is available only as a Windows executable. We were, however, able to experiment with two datasets. The first dataset is the Scholl dataset that was used to demonstrate IQTest in the paper [10]. The second dataset we tested is the Seidl dataset from QBFLIB.

A plot comparing the runtimes for the Scholl and Seidl datasets is shown in Figure 3. The experiments were run on a 2.41GHz machines with 2GB of RAM. An instance is plotted with the time CirQit took to solve it on the x-axis and IQTest on the y-axis. So, an instance above the bisecting line is one on which our solver exhibited superior performance, and an instance below the line is one where IQTest was superior. Timed out instances are placed at the 1200 second mark on the graph.

On the first dataset, IQTest outperforms CirQit. There are eight problems that IQTest was able to solve, sometimes fairly quickly, but CirQit was unable solve in the time allotted. However, there were also a number of problems that CirQit was able to solve a few orders of magnitude faster than IQTest. On the Seidl dataset, on the other hand, CirQit confidently outperforms IQTest. The detailed results were that on Scholl, containing 63 problems, CirQit solved 38 problems in 382 seconds, while IQTest solved 46 problems in 3,887 seconds. On the other hand, on Seidl, containing 150 problems, CirQit solved 147 problems in 2,969 seconds, while IQTest solved 126 problems in 53,110 seconds.

Although this is not a complete analysis, these sets show that while IQTest is better than CirQit on some problems, there are problem suites for which CirQit is better suited.

Finally, although we don't show any results, we did experiment with CirQit turning don't care propagation on and off. Over a large number of problems we found that don't care propagation yielded on average almost a 40% speedup.

6 Conclusions and Future Work

This paper demonstrates the effectiveness of exploiting structural information in QBF solving. By skipping the last step of encoding QBF problems into CNF, the structure of the problem can be maintained and used by a DPLL search engine. While other work has been done in the past to overcome some of the limitations of the CNF representation, our circuit based solver includes many of the benefits realized by these partial solutions.

We demonstrated that a solver using a circuit representation can be highly competitive with state of the art solvers using both non-CNF and CNF representations.

The circuit representation is compact, and allows more powerful propagation. Many more benefits could potentially be reaped from the circuit representation. The circuit representation allows the solver to generate CNF clauses for clause learning on-the-fly. We believe that it is possible to use the circuit in a similar way to extract DNF cubes on the fly. We are investigating this approach.

Many orthogonal improvements, such as exploiting non-prenex structure or using problem decomposition, can also be applied to the circuit solver.

In sum it seems that using a circuit representation is a very fruitful direction for obtaining further advances in QBF solving.

References

1. Rintanen, J.: Asymptotically optimal encodings of conformant planning in QBF. In: Proceedings of the AAAI National Conference (AAAI), pp. 1045–1050 (2007)
2. Egly, U., Eiter, T., Tompits, H., Woltran, S.: Solving advanced reasoning tasks using quantified boolean formulas. In: Proceedings of the AAAI National Conference (AAAI), pp. 417–422. AAAI Press, Menlo Park (2000)
3. Mangassarian, H., Veneris, A.G., Safarpour, S., Benedetti, M., Smith, D.: A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test. In: International Conference on Computer-Aided Design (ICCAD), pp. 240–245 (2007)
4. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7, 201–215 (1960)
5. Biere, A.: Resolve and expand. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004, vol. 3542, pp. 59–70. Springer, Heidelberg (2005)
6. Benedetti, M.: sKizzo: a QBF decision procedure based on propositional skolemization and symbolic reasoning. Technical Report TR04-11-03 (2004)
7. Thiffault, C., Bacchus, F., Walsh, T.: Solving non-clausal formulas with DPLL search. In: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT) (2004)
8. Wu, C.A., Lin, T.H., Lee, C.C., Huang, C.Y.: QuteSAT: a robust circuit-based SAT solver for complex circuit structure. In: Design, Automation and Test in Europe Conference and Exposition (DATE), pp. 1313–1318 (2007)
9. Sabharwal, A., Ansótegui, C., Gomes, C.P., Hart, J.W., Selman, B.: QBF modeling: Exploiting player symmetry for simplicity and efficiency. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 382–395. Springer, Heidelberg (2006)
10. Zhang, L.: Solving QBF with combined conjunctive and disjunctive normal form. In: Proceedings of the AAAI National Conference (AAAI) (2006)

11. Tseitin, G.: On the complexity of proofs in poropositional logics. In: Siekmann, J., Wrightson, G. (eds.) Automation of Reasoning: Classical Papers in Computational Logic 1967–1970, vol. 2. Springer, Heidelberg (1983); Originally published (1970)
12. Zhang, L., Malik, S.: Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In: Van Hentenryck, P. (ed.) CP 2002, vol. 2470, pp. 200–215. Springer, Heidelberg (2002)
13. Tang, D., Malik, S.: Solving quantified boolean formulas with circuit observability don't cares. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 368–381. Springer, Heidelberg (2006)
14. Benedetti, M., Lallouet, A., Vautard, J.: QCSP made practical by virtue of restricted quantification. In: Veloso, M.M. (ed.) Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 38–43 (2007)
15. Egly, U., Seidl, M., Woltran, S.: A solver for QBFs in negation normal form. Constraints 14(1), 38–79 (2009)
16. Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantified Boolean Formulas satisfiability library (QBFLIB) (2001), www.qbflib.org
17. Stéphan, I.: Boolean propagation based on literals for quantified boolean formulae. In: 17th European Conference on Artificial Intelligence (2006)
18. Benedetti, M.: skizzo: A suite to evaluate and certify QBFs. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 369–376. Springer, Heidelberg (2005)
19. Samulowitz, H., Bacchus, F.: Dynamically partitioning for solving QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 215–229. Springer, Heidelberg (2007)
20. Giunchiglia, E., Narizzano, M., Tacchella, A.: QUBE: A system for deciding Quantified Boolean Formulas satisfiability. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 364–369. Springer, Heidelberg (2001)

Solving (Weighted) Partial MaxSAT through Satisfiability Testing*

Carlos Ansótegui[1], María Luisa Bonet[2], and Jordi Levy[3]

[1] Universitat de Lleida (DIEI, UdL)
[2] Universitat Politècnica de Catalunya (LSI, UPC)
[3] Artificial Intelligence Research Institute (IIIA, CSIC)

Abstract. Recently, Fu and Malik described an unweighted Partial MaxSAT solver based on successive calls to a SAT solver. At the kth iteration the SAT solver tries to certify that there exist an assignment that satisfies all but k clauses. Later Marques-Silva and Planes implemented and extended these ideas. In this paper we present and implement two Partial MaxSAT solvers and the weighted variant of one of them. Both are based on Fu and Malik ideas. We prove the correctness of our algorithm and compare our solver with other (Weighted) MaxSAT and (Weighted) Partial MaxSAT solvers.

1 Introduction

In real-life, some solutions to a problem are acceptable even when some constraints are violated. In fact, in many situations it is impossible to satisfy all constraints. For instance, in the context of planning, scheduling, packing, etc., a solution satisfying all the constraints may be impossible to obtain. However we are still interested on which is the maximum number of constraints that can be satisfied with a minimal penalty.

We can solve these problems through the use of MaxSAT formalisms, such as (Weighted) MaxSAT and (Weighted) Partial MaxSAT. Recently, there has been an increasing interest in the development of solvers for these formalisms. Since 2006, every year takes place the MaxSAT evaluation [2]. Most of the solvers submitted to the last MaxSAT08 evaluation are implementations of branch&bound algorithms (MaxSatz [11], IncWMaxSatz, W-MaxSatz, WMaxsatz_icss [6], MiniMaxSat [10], Lb-Sat and Lb-PSat [12,13], PMS [3], ToolBar3 [9]). There are other approaches like the solver Clone [17], that makes use of a tractable language known as d-DNNF, and those which are based on the use of Satisfiability testing, SAT4J [4], msu1.2 [14,15] and msu4.0 [16].

None of these solvers is a clear winner, specially for industrial and crafted instances. In particular, for the industrial category the solvers based on Satisfiability testing seem to perform very well for many benchmarks. Since the ultimate goal is to solve real world (industrial) instances it makes sense to study in detail

* Research partially supported by the projects TIN2007-68005-C04-{01,03,04} and TIN2006-15662-C02-02 funded by the MEC.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 427–440, 2009.

this approach. Why these solvers work better for industrial instances may be a phenomena not only related to the hardness of the unsatisfiability cores included in the formulas (that can be efficiently detected by a SAT solver) but also to how these cores are connected.

The base of the study of this paper is the work of Fu and Malik [7,8], where two Partial MaxSAT algorithms based on calls to a SAT solver are proposed, and the work of Marques-Silva and Planes [15,16] and Marques-Silva and Manquinho [14] which extend that previous work.

The contributions of our work are (i) a more optimized implementation of the original Fu and Malik algorithm; (ii) a weighted version of the original Fu and Malik algorithm together which the proof of its correctness; and (iii) another Partial MaxSAT solver variant of the Fu and Malik algorithm, and the proof of its correctness.

For the purpose of the evaluation of these algorithms, there is only one solver, SAT4J [4], that is adapted to deal with weights. In this paper, we provide a weighted version of the Fu and Malik algorithm [8]. Our experimental investigation confirms what we already knew from previous MaxSAT evaluations. There is no unique best algorithm for solving MaxSAT or the other variants. Nevertheless, our implementation has a better performance than other solvers based on Satisfiability testing. In the case of the Partial Weighted MaxSAT, our solver is the first implementation of the original Fu and Malik ideas extended to the weighted problem. Therefore, we can only compare with SAT4J.

2 Preliminaries

In the Partial MaxSAT context we work with two sets of clauses, *hard* and *soft*. The *Partial MaxSAT problem* for a multiset of clauses is the problem of finding an *optimal assignment* to the variables that satisfies all the hard clauses, and the maximum number of soft clauses. The number of soft clause falsified by an assignment is the *cost* of this assignment. The cost of the optimal assignment of a formula F is called the cost of the formula, and is denoted by $MaxSAT(F)$.

In Weighted Partial MaxSAT, we use multisets of *weighted clauses*. A weighted clause is a pair (C, w), where C is a clause and w is a natural number meaning the penalty for falsifying the clause C. The pair (C, w) is clearly equivalent to having w copies of clause C in our multiset (in case C is soft). If a clause is hard, the corresponding weight is infinity.

Given a truth assignment I and a multiset of weighted clauses $\mathcal{C}$, the *cost* of assignment I on $\mathcal{C}$ is the sum of the weights of the clauses falsified by I.

The *Weighted Partial MaxSAT problem* for a multiset of weighted clauses $\mathcal{C}$ is the problem of finding an *optimal assignment* to the variables of $\mathcal{C}$ that minimizes the cost of the assignment on $\mathcal{C}$. If the cost is infinity, it means that we have falsified a hard clause, and we say that the multiset is *unsatisfiable*.

Our approach is based on successive calls to a SAT solver. The SAT solver may return a set of clauses that is unsatisfiable. We call this set *unsatisfiable core*.

3 A Weighted Partial MaxSAT Algorithm

Before giving the full version of our algorithm, we will present the original Fu and Malik [8] algorithm for Partial MaxSAT, and show the correction of the algorithm. The reason for doing this is that we will need parts of the argument to show the correctness of our algorithm for solving Weighted Partial MaxSAT.

The algorithm consists in iteratively calling a SAT solver on a working formula φ. This corresponds to the line $(st, \varphi_c) := SAT(\varphi_w)$. The SAT solver will say whether the formula is satisfiable or not (variable st), and in case the formula is unsatisfiable, it will give an unsatisfiable core (φ_c). At this point the algorithm will produce new variables, blocking variables (BV in the code), one for each clause. The new working formula φ will consist in adding the new variables to the formulas of the core, adding a cardinality constraint saying that exactly one of the new variables should be true ($CNF(\sum_{b \in BV} b = 1)$ in the code), and adding one to the counter of falsified clauses. This procedure is applied until the SAT solver returns satisfiable.

```
input: φ = {C_1, ..., C_m}
cost := 0                                   Optimal
while true do
    (st, φ_c) := SAT(φ)                     Call to the SAT solver
    if st = SAT then return cost
    BV := ∅                                 Set of blocking variables
    for each C ∈ φ_c do
        if C is soft then
            b := new blocking variable
            φ := φ \ {C} ∪ {C ∨ b}          Add blocking variable
            BV := BV ∪ {b}
    if BV = ∅ then return UNSAT             There are no soft clauses in the core
    φ := φ ∪ CNF(Σ_{b∈BV} b = 1)            Add cardinality constraint as hard clauses
    cost := cost + 1
```

Fig. 1. The pseudo-code of the Fu&Malik algorithm

The following lemma and definition are part of the correctness of the algorithm for both the weighted and unweighted versions.

Definition 1. *We say that two (Weighted) (Partial) MaxSAT formulas φ and φ' are MaxSAT equivalent if the cost of the optimal assignment of φ is equal to the cost of the optimal assignment of φ'.*

Lemma 1. *Let φ be an unsatisfiable CNF formula, and $\varphi_c = \{C_1, \ldots, C_s\}$ be an unsatisfiable core in φ. Define*

$$\varphi' = (\varphi \setminus \varphi_c) \ \cup \ \{C_1 \vee b_1, \ldots, C_s \vee b_s\} \cup \ CNF(\sum_{i=1}^{s} b_i = 1) \ \cup \ \{\square\}$$

where $b_1, \ldots, b_s$ are new variables.

Then, the minimum number of falsified clauses of φ is the same as the minimum number of falsified clauses of φ', i.e. φ and φ' are MaxSAT equivalent.

PROOF: Let I be a truth assignment for the variables of φ that satisfies all the hard clauses of φ. Since φ_c is an unsatisfiable core, I falsifies some clause in φ_c. Let C_i be one such clause. Now define I' the following way: for all $x \in \varphi$, $I'(x) = I(x)$; $I'(b_i) = 1$ and $I'(b_j) = 0$ for all $j \neq i$, $1 \leq j \leq s$. Now I' satisfies $CNF(\sum_{i=1}^{s} b_i = 1)$. For every clause C in $\varphi \setminus \varphi_c$, $I'(C) = I(C)$, and the same is true for all the clauses $C_j \vee b_j$ for $j \neq i$. Now, I falsifies C_i but I' satisfies $C_i \vee b_i$ and falsifies $\square$. As a consequence the number of falsified clauses of φ' by I' is the same as the number of falsified clauses of φ by I.

Now consider an optimal assignment I' for φ'. By the optimality of I', we know that I' satisfies $CNF(\sum_{i=1}^{s} b_i = 1)$ and if $I'(C_i) = 1$ then $I'(b_i) = 0$. Now we define an assignment I for φ the following way: $I(x) = I'(x)$ for all $x \in \varphi$. We will see that the number of falsified clauses of I is the same as the number of falsified clauses in I'. The number of falsified clauses in $\varphi \setminus \varphi_c$ is clearly the same. Let b_i be the variable assigned true by I'. Then $I'(C_j \vee b_j) = I(C_j)$ for $j \neq i$. On the other hand, $I(C_i) = 0$ and $I'(C_i \vee b_i) = 1$ but I' falsifies $\square$. ■

Theorem 1. FU&MALIK *is a correct algorithm for Partial MaxSAT.*

PROOF: In each iteration of the while loop, if the SAT solver returns unsatisfiable and the unsatisfiable core has soft clauses, we substitute a formula φ by another φ' plus the addition of one to the variable *cost*. Adding 1 to *cost* is equivalent to considering that φ' has also the empty clause. Lemma 1 shows that both formulas are equivalent in terms of the minimum number of unsatisfiable clauses. ■

The following algorithm is the weighted version of the previous one. Now we iteratively call the SAT solver with the working formula without the weights. When the SAT solver returns an unsatisfiable core, we calculate the minimum weight of the clauses of the core, w_{min} in the algorithm. Now we transform the working formula in the following way: we duplicate the core having on one of the copies, the clauses with weight the original minus the minimum weight, and on the other copy we put the blocking variables and we give it the minimum weight. Finally we add the cardinality constraint on the blocking variables, and we add w_{min} to the *cost*.

Lemma 2. *Let φ be a weighted partial formula. Let $exp(\varphi)$ be the natural expansion of φ into an unweighted formula by substituting every clause (C, w) of φ into w copies of C.*

The minimum weight of φ is the same as the minimum number of falsified clauses of $exp(\varphi)$.

PROOF: This is straightforward. ■

The next lemma shows that if we have several identical unsatisfiable cores, we don't need to add different blocking variables to each core. Instead all cores can have the same set of blocking variables.

input: $\varphi = \{(C_1, w_1), \ldots, (C_m, w_m)\}$
$cost := 0$ — Optimal
while *true* **do**
  $(st, \varphi_c) := SAT(\{C_i \mid (C_i, w_i) \in \varphi\})$ — Call to the SAT solver without weights
  if $st = SAT$ **then return** $cost$
  $BV := \emptyset$ — Blocking variables of the core
  $w_{min} := \min\{w_i \mid C_i \in \varphi_c \text{ and } C_i \text{ is soft}\}$
  for each $C_i \in \varphi_c$ **do**
    if C_i is soft **then**
      $b_i :=$ new blocking variable
      $\varphi := \varphi \setminus \{(C_i, w_i)\} \cup \{(C_i, w_i - w_{min})\} \cup \{(C_i \vee b_i, w_{min})\}$
      Duplicate soft clauses of the core
      $BV := BV \cup \{b_i\}$
  if $BV = \emptyset$ **then return** UNSAT — There are no soft clauses in the core
  else $\varphi := \varphi \cup CNF(\sum_{b \in BV} b = 1)$ — Add cardinality constraint as hard clauses
  $cost := cost + w_{min}$

Fig. 2. The pseudo-code of the WPM1 algorithm

Lemma 3. *Let φ be an unsatisfiable partial formula and let $\varphi_c = \{C_1, \ldots, C_s\}$ be an unsatisfiable core in φ that appears l times. Consider the following formulas:*

$$\varphi_1 = \varphi \setminus \varphi_c \cup \{\underbrace{C_i \vee b_i, \ldots, C_i \vee b_i}_{l \text{ times}} \mid C_i \in \varphi_c\} \cup CNF(\sum_{i=1}^{s} b_i = 1)$$

and

$$\varphi_2 = \varphi \setminus \varphi_c \cup \{C_i \vee b_i^1, \ldots, C_i \vee b_i^l \mid C_i \in \varphi_c\} \cup CNF(\textstyle\sum_{i=1}^{s} b_i^1 = 1) \cup \ldots \cup CNF(\sum_{i=1}^{s} b_i^l = 1)$$

Then, the minimum number of unsatisfiable clauses of φ_1 and φ_2 are the same, i.e. φ_1 and φ_2 are MaxSAT equivalent.

Proof: Let I be an optimal interpretation for φ_1. Then, if $I(C_i) = 1$, for some $i = 1, \ldots, s$, then $I(b_i) = 0$. This is true because φ_c is unsatisfiable and I is optimal. Now we will modify I into an assignment I' the following way:

$$I'(x) = I(x) \text{ for all } x \in \varphi$$
$$I'(b_i^j) = I(b_i) \text{ for } i = 1, \ldots, s \text{ and } j = 1, \ldots, l$$

It is clear that for all $C \in \varphi - \varphi_c$, $I'(C) = I(C)$. Also, $I'(C_i \vee b_i^j) = I(C_i \vee b_i)$.

Let now I' be an optimal assignment for φ_2. Then as before, if $I(C_i) = 1$, for some $i = 1, \ldots, s$, then $I(b_i^j) = 0$ for every $j = 1, \ldots, l$. Now we will modify I' into an assignment I the following way:

$$I(x) = I'(x) \text{ for all } x \in \varphi$$
$$I(b_i) = I(b_i^1) \text{ for all } i = 1, \ldots, s$$

It is clear that for all $C \in \varphi - \varphi_c$, $I'(C) = I(C)$. We will show that the number of unsatisfied clauses of $\{C_1 \vee b_1^j, \ldots, C_s \vee b_s^j\}$ by I' for every $j = 1, \ldots, l$ is the same as the number of unsatisfied clauses of $\{C_i \vee b_1, \ldots, C_s \vee b_s\}$ by I. Now suppose that the only b variable that I assigns true is b_i^j and the only b variable that I assigns true is b_k. By assumption, $I(C_i) = I(C_k) = 0$. Then $I'(C_i \vee b_i^j) = 1$ and $I'(C_k \vee b_k^j) = 0$, but $I(C_i \vee b_i) = 0$ and $I(C_k \vee b_k) = 1$. ■

The next lemma shows the correctness of one iteration of our Weighted Partial MaxSAT algorithm WPM1.

Lemma 4. *Let φ be an unsatisfiable weighted partial formula, let $\varphi_c = \{C_1, \ldots, C_s\}$ be an unsatisfiable core in the set of clauses from φ, and let $\varphi_c^w = \{(C_1, w_1), \ldots, (C_s, w_s)\}$ the subset of weighted clauses of φ that corresponds to the core. Let $w_{min} = \min(w_1, \ldots, w_s)$, and let*

$$\begin{aligned}\varphi' =& (\varphi \setminus \varphi_c^w) \cup \{(C_i, w_i - w_{min}) \mid C_i \in \varphi_c\} \\ & \cup \{(C_i \vee b_i, w_{min}) \mid C_i \in \varphi_c\} \\ & \cup \mathit{CNF}(\textstyle\sum_{i=1}^{s} b_i = 1) \cup \{(\square, w_{min})\}\end{aligned}$$

where $\{b_1, \ldots, b_s\}$ is a set of new variables.
Then, φ and φ' are MaxSAT equivalent.

Proof: Let $exp(\varphi)$ be the unweighted expansion of φ. Lemma 2 shows that the minimum weight of φ is the same as the number of falsified clauses of $exp(\varphi)$. Now $\varphi_c = \{C_1, \ldots, C_s\}$ is an unsatisfiable core of $exp(\varphi)$, and since $w_{min} = \min(w_1, \ldots, w_s)$, φ_c appears w_{min} times in $exp(\varphi)$. Now we can apply the transformation of lemma 1 w_{min} times to obtain a formula

$$\begin{aligned}\varphi_2 =& \varphi \setminus \varphi_c \cup \{C_i \vee b_i^1 \ldots C_i \vee b_i^{w_{min}} \mid C_i \in \varphi_c\} \\ & \cup \mathit{CNF}(\textstyle\sum_{i=1}^{s} b_i^1 = 1) \cup \ldots \cup \mathit{CNF}(\sum_{i=1}^{s} b_i^l = 1) \\ & \{ \underbrace{C_i, \ldots, C_i}_{w_i - w_{min} \text{ copies}} \mid C_i \in \varphi_c\} \cup \{\underbrace{\square, \ldots, \square}_{w_{min}}\}\end{aligned}$$

MaxSAT equivalent to φ_{exp}. By Lemma 3, φ' is MaxSAT equivalent to the formula

$$\begin{aligned}\varphi_1 =& \varphi \setminus \varphi_c \cup \{\underbrace{C_i \vee b_i, \ldots, C_i \vee b_i}_{w_{min} \text{ copies}} \mid C_i \in \varphi_c\} \\ & \cup \{ \underbrace{C_i, \ldots, C_i}_{w_i - w_{min} \text{ copies}} \mid C_i \in \varphi_c\} \\ & \cup \mathit{CNF}(\textstyle\sum_{i=1}^{s} b_i = 1) \cup \{\underbrace{\square, \ldots, \square}_{w_{min}}\}\end{aligned}$$

Now using again Lemma 2, φ_1 is MaxSAT equivalent to φ' as in the statement of the lemma. ■

Theorem 2. *WPM1 is a correct algorithm for Weighted Partial MaxSAT.*

PROOF: The theorem is proved iterating Lemma 4 for every execution of the loop of the algorithm. ■

4 Another Partial MaxSAT Algorithm

The next algorithm, that we call PM2, is also a variant of the Fu and Malik algorithm that avoids the use of more than one blocking variable in a clause. A single blocking variable is added to each soft clause, like in other solvers like SAT4J [4], msu3 [15] and msu4.0 [16].

PM2 works as follows: every clause gets an additional variable and the cardinality constraint says that all these additional variables have to be false. Also before the first iteration of the algorithm the counter of falsified clauses, *cost*, is set to zero. At every iteration of the algorithm a SAT solver is called. As before, if the solver returns unsatisfiable, it also gives an unsatisfiable core. If the core only contains hard clauses, then the algorithm returns unsatisfiable. Otherwise, we put the blocking variables of the soft clauses of the core in a set B. Since we have found a new unsatisfiable core, variable *cost* gets increased by one. Also we look for other cores such that the soft clauses are included in the new core. If no such core exists, we add an *at least* cardinality constraint saying that the sum of the blocking variables of B is larger than or equal to one. If some cores are included, we add the cardinality constraint saying that the number of variables in B that need to be one is at least the number of cores included in the last core found (counting the last). In every call to the SAT solver we also add an *at most* cardinality constraint saying that the sum of all blocking variables is at most *cost*. If the solver says that the formula is satisfiable, the algorithm returns *cost* as the minimal number of falsified clauses.

PM2 simplifies FU&MALIK in the sense that it only adds one blocking variable per clause. Intuitively, this would have to result into a more efficient algorithm because there are less blocking variables, so the SAT solver will have to check less possible assignments. This idea is already used in other MaxSAT solvers, like SAT4J [4], msu3 [15] and msu4.0 [16]. In SAT4J only one *at most* cardinality constraint (saying that the sum of blocking variables is smaller than k) is used. This bound k is reduced until the SAT solvers says unsatisfiable. In msu3 [15], in a first phase they compute a maximal set of disjoint cores, and in a second phase they do as in SAT4J but increasing the bound k (starting with the number of disjoint cores) until the SAT solver returns sat, and only summing the blocking variables that have appeared in some core. Finally, in the msu4.0 algorithm [16], apart from the *at most* constraint, they also use some *at least* constraints saying that blocking variables occurring in a core, and not occurring in previous cores, have to sum at least one. The algorithm alternates phases where the SAT solver returns sat or unsat, refining a lower or upper bound, and only terminates when

the upper and lower bound coincide, or when the new core does not contain new blocking variables. Our approach is different from previous ones in two senses. First, our *at most* constraint has a bound *cost* that is successively increased like in msu3, instead of decreased like in SAT4J. Second, our *at least* constraints may impose a bound strictly greater than one, in contrast with the msu4.0 algorithm. This would have to result in a more restrictive constraint, thus in fewer assignments to check by the SAT solver.

input: $\varphi = \{C_1, \ldots, C_m\}$
$BV := \{b_1, \ldots, b_m\}$ — Set of all blocking variables
$\varphi_w := \{C_1 \vee b_1, \ldots, C_m \vee b_m\}$ — Protect all clauses
$cost := 0$ — Optimal
$L := \emptyset$ — Set of Cores
while *true* **do**
  $(st, \varphi_c) := SAT(\varphi_w \cup CNF(\sum_{b \in BV} b \leq cost))$ — Call to the SAT solver with at most cardinality constraint
  if $st = SAT$ **then return** *cost*
  remove the hard clauses from φ_c
  if $\varphi_c = \emptyset$ **then return** UNSAT
  $B := \emptyset$ — Blocking variables of the core
  for each $C = C_i \vee b_i \in \varphi_c$ **do**
    $B := B \cup \{b_i\}$
  $L := L \cup \{\varphi_c\}$
  $k := |\{\psi \in L \mid \psi \subseteq \varphi_c\}|$ — Num. of cores contained in φ_c including φ_c
  $\varphi_w := \varphi_w \cup CNF(\sum_{b \in B} b \geq k)$ — Add at least cardinality constraint
  $cost := cost + 1$

Fig. 3. The pseudo-code of the PM2 algorithm

To prove that the PM2 algorithm is correct, we will prove that the FU&MALIK algorithm can simulate it. We have to be aware they are non-deterministic, since we assume that the SAT solver returns an unsatisfiable core non-deterministically. However, recall that we have proved that FU&MALIK algorithm is correct for every possible run. The proof is by induction on the number of execution steps. From now on, when we say that a set of soft clauses is a core, we mean that this set, together with the hard clauses, is a core. A core $B = \{i_1, \ldots, i_m\}$ will be a set of indexes of soft clauses. Suppose that FU&MALIK has simulated PM2 for s steps. We will prove that 1) if PM2 finds a core B, then this set B of (soft) clauses is also a core for the FU&MALIK algorithm, in particular if the set $B = \emptyset$ is a core for PM2, and it returns UNSAT, then the same set $B = \emptyset$ is a core for FU&MALIK, that also returns UNSAT; and 2) if PM2 does not find any cores, and stops, then FU&MALIK does not find any cores either, and also stops returning the same MaxSAT value, since both have run the same number of steps.

Since Theorem 3 will be proved by induction, assume by induction hypothesis that

$$\varphi = \{C_1 \vee a_1, \ldots, C_m \vee a_m\} \cup CNF(\sum_{i=1}^{m} a_i \leq s) \cup \bigcup_{r=1}^{s} CNF(\sum_{i \in B_r} a_i \geq k_r)$$

is the formula computed by PM2 after s execution steps, where $B_1, \ldots, B_s$ is the sequence of cores, and k_j is the number of cores from $B_1, \ldots, B_j$ included in B_j. Assume also that FU&MALIK, after s steps simulating PM2, with the same sequence of cores $B_1, \ldots, B_s$, obtains the formula

$$\hat{\varphi} = \{C_1 \vee \bigvee_{1 \in B_j} b_1^j, \ldots, C_n \vee \bigvee_{n \in B_j} b_n^j\} \cup \bigcup_{j=1}^{s} CNF(\sum_{i \in B_j} b_i^j = 1)$$

Lemma 5. *Let φ and $\hat{\varphi}$ be the formulas obtained by* PM2 *and* FU&MALIK *algorithms, respectively, after s steps of simulation. For any optimal interpretation $\hat{I}$ of the variables of $\hat{\varphi}$, let I be the interpretation of the variables of φ given by*

$$\begin{array}{ll} I(x) = \hat{I}(x) & \text{for any variable } x \in \{C_1, \ldots, C_n\} \\ I(a_i) = \max\{\hat{I}(b_i^j) \mid i \in B_j\} & \text{for the blocking variables} \end{array}$$

Then, (1) if $\hat{I}$ satisfies the hard clause $C \in \hat{\varphi}$, then I satisfies the hard clause $C \in \varphi$;
(2) if $\hat{I}$ satisfies the cardinality constraints of $\hat{\varphi}$, then I satisfies the cardinality constraints of φ; and
(3) if $\hat{I}$ satisfies the soft clause $C_i \vee \bigvee_{i \in B_j} b_i^j \in \hat{\varphi}$, then I satisfies the soft clause $C_i \vee a_i \in \varphi$.

PROOF: Statement (1) is trivial, since I and $\hat{I}$ assign the same values to the original variables. For (2), if $\hat{I}$ satisfies the cardinality constraints of $\hat{\varphi}$, then $\sum_{i \in B_r} \hat{I}(b_i^r) = 1$, for any core B_r. Hence,

$$\sum_{i \in B_r} \sum_{\substack{i \in B_j \\ j=1,\ldots s}} \hat{I}(b_i^j) \geq \sum_{\substack{B_j \subset B_r \\ j \leq r}} \sum_{i \in B_j} \hat{I}(b_i^j) = |\{B_j \mid B_j \subseteq B_r \wedge j \leq r\}|$$

for any of the cores B_r obtained in the execution. If the interpretation $\hat{I}$ is optimal, it means that it assigns *true* to at most one of the blocking variables of a clause. In other words,

$$\sum_{\substack{i \in B_j \\ j=1,\ldots,s}} \hat{I}(b_i^j) \leq 1 \quad \text{hence} \quad I(a_i) = \max\{\hat{I}(b_i^j) \mid i \in B_j \wedge j = 1, \ldots, s\} = \sum_{\substack{i \in B_j \\ j=1,\ldots,s}} \hat{I}(b_i^j)$$

for any $i = 1, \ldots, n$. From all this, we conclude $\sum_{i \in B_r} I(a_i) \geq |\{B_j \mid B_j \subseteq B_r \wedge j \leq r\}| = k_r$, i.e. I satisfies the cardinality constraints $CNF(\sum_{i \in B_r} a_i \geq k_r)$.

Similarly, we can prove that if $\hat{I}$ is optimal

$$\sum_{i=1}^{n} I(a_i) = \sum_{i=1}^{n} \max\{\hat{I}(b_i^j) \mid i \in B_j \wedge j = 1, \ldots, s\}$$
$$= \sum_{i=1}^{n} \sum_{\substack{i \in B_j \\ j=1,\ldots,s}} \hat{I}(b_i^j) = \sum_{j=1}^{s} \sum_{i \in B_j} \hat{I}(b_i^j) = \sum_{j=1}^{s} 1 = s$$

Hence, the other cardinality constraint $CNF(\sum_{i=1}^{m} a_i \leq s)$ is also satisfied.

For (3), if $\hat{I}$ satisfies $C_i \vee \bigvee_{i \in B_j} b_i^j$, then either it satisfies C_i, and so I because the assign the same values to original variables, or it satisfies some of the variables b_i^j. In this second case, the way we define the value of a_i ensures that I satisfies this value, hence the clause $C_i \vee a_i \in \varphi$. ■

Lemma 6. *Let φ and $\hat{\varphi}$ be the formulas obtained by* PM2 *and* Fu&Malik *algorithms, respectively, after s steps of simulation. If B is a core of φ, then B is also a core of $\hat{\varphi}$.*

Proof: If B is not a core of $\hat{\varphi}$, then there exists an interpretation I of $\hat{\varphi}$ that satisfies all hard clauses, all cardinality constraints of $\hat{\varphi}$, and all soft clauses $C_i \vee \bigvee_{i \in B_j} b_i^j$ where $i \in B$. By Lemma 5, I will satisfy all hard clauses, all cardinality constraints of φ, and all soft clauses $C_i \vee a_i$ where $i \in B$. This would contradict that B is a core of φ. ■

The previous lemma ensures that if PM2 finds a core, then Fu&Malik also finds a core. Hence, if PM2 does not stop, then Fu&Malik does not stop, either. Therefore, the values computed by PM2 are smaller than the values calculated by Fu&Malik. However, it is still possible that PM2 computes underestimated values of MaxSAT of a formula. The following lemma shows that this is not the case. Notice that the proof of this lemma relies on the correctness of the Fu&Malik algorithm.

Lemma 7. *Let φ and $\hat{\varphi}$ be the formulas obtained by* PM2 *and* Fu&Malik *algorithms, respectively, after s steps of simulation. If φ is satisfiable, then $\hat{\varphi}$ is satisfiable.*

Proof: Let I be an interpretation satisfying φ. In particular, I satisfies $CNF(\sum_{i=1}^{m} a_i \leq s)$, and all hard and soft clauses. Therefore, I satisfies all the *original* soft clauses C_i where $I(a_i) = false$, and there are at least $n - s$ of such clauses. We have $MaxSAT(C_1, \ldots, C_n) \leq s$. Since, Fu&Malik is a correct algorithm for MaxSAT, it has to stop before s or less execution steps. And, since it has not finished before, it has to finish after these s steps, hence $\hat{\varphi}$ is satisfiable. ■

Theorem 3. PM2 *is a correct algorithm for Partial MaxSAT.*

Table 1. Time in seconds (solved). Timeout of 1200 seconds. # stands for number of instances of the benchmark.

set	#	best08	WPM1	PM2	msu1.2	msu4.0	SAT4J
Unweighted MaxSAT Category							
Crafted							
Maxcut/dimacs_mod/	62	IncMaxSatz - **81.8(52)**	0.03(4)	175(7)	0.28(4)	1.71(3)	0.93(2)
Maxcut/random/	58	MaxSatz - **4.5(40)**	-(0)	-(0)	- (0)	- (0)	- (0)
Maxcut/Spinglass/	5	MiniMaxSatz - **1.62(3)**	0.85(2)	102.5(2)	0.68 (2)	-(0)	-(0)
Industrial							
SeanSafarpour	112	msu1.2 - 57.5(72)	**66.6(81)**	90.2(75)	57.5(72)	64.4(50)	14.5(10)
Partial MaxSAT Category							
Crafted							
Maxclique/Random/	96	MiniMaxSAT - **2.4(96)**	50.4(1)	-(0)	-(0)	106(61)	114(52)
Maxclique/Structured/	62	MiniMaxSAT - **73(36)**	41.2(11)	32.6(6)	4.9(7)	105.2(13)	50.5(13)
Maxone/3SAT/	80	IncMaxSatz - **0.46(80)**	15.82(46)	105.7(79)	52.7(40)	118.2(35)	96.6(31)
Maxone/Structured/	60	SAT4J - **10.1(60)**	0.69(2)	547.5(13)	122.7(2)	3.34(1)	**10.1(60)**
Industrial							
Bcp-fir/	59	msu1.2 - 49.2(46)	**31.7 (57)**	67.4(56)	49.2(46)	-(0)	13.3(10)
Bcp-hipp-yRa1/	1183	SAT4J - 19.2(1111)	2.9(1122)	**13.5(1162)**	7.2(1105)	0.29(348)	12.20(1109)
Bcp-msp/	148	MiniMaxSAT - **48.9(104)**	15.5(26)	384.2(36)	4.9(25)	22.9(79)	8.8(93)
Bcp-mtg/	215	MiniMaxSAT - 25.7(206)	5.8(170)	**10.5(214)**	17.5(164)	0.43(22)	57(196)
Bcp-syn/	74	lb-psat - **63.4(34)**	14.1(32)	**71.2(34)**	51.1(31)	105.2(11)	67.4(21)
Pbo-mqc-nencdr/	128	msu4.0 - **167.5(115)**	80.4(50)	142(78)	50.3(54)	**167.5(115)**	180.6(102)
Pbo-mqc-nlogencdr/	128	msu4.0 - **111(128)**	67.1(75)	140.3(97)	53(65)	**111(128)**	117.5(126)
Pbo-routing/	15	msu1.2 - 2.9(15)	**0.94(15)**	24.7(15)	2.9(15)	54.9(15)	26.4(9)

5 Experimental Results

In order to conduct our experimental investigation we have selected the benchmarks submitted to the MaxSAT08 evaluation [2]. We have focus on the crafted and industrial instances for all the four categories: unweighted MaxSAT, partial MaxSAT, weighted MaxSAT and weighted partial MaxSAT. The appropriate testing instances for our algorithms would be the industrial instances, where we can expect these approaches to be competitive. However, since there is a lack of industrial instances, in particular for the weighted and partial weighted categories, we decided also to incorporate the crafted instances.

Our experiments have been run on the same machine specs as the MaxSAT evaluation; Operating System: Rocks Cluster 4.0.0 Linux 2.6.9, Processor: AMD Opteron 248 Processor, 2 GHz and compilers, Memory: 1 GB and Compilers GCC 3.4.3, javac JDK 1.5.0. The solvers we compare are the best solvers for each category and benchmark at the MaxSAT08 evaluation [2], the solvers based on satisfiability testing (msu1.2, msu4.0 [16], and SAT4J [4]) and our implementations of the weighted version of Fu&Malik (WPM1) and Partial MaxSAT 2 (PM2) presented in this paper.

Our solvers are implemented on top of the SAT solver picosat846 [5], although they can be easily adapted to work with any other solver that provides an

Table 2. Time in seconds (solved). Timeout of 1200 seconds. # stands for number of instances of the benchmark.

set	#	best08	WPM1	SAT4J
Weighted MaxSAT Category				
Crafted				
KeXu/	15	IncWMaxsatz - **126.5(15)**	478(1)	7.7(4)
Ramsey/	48	lb-psat - **1.63(37)**	0.05(34)	16(35)
WMaxcut/dimacs_mod/	62	ToolBar3 - **59(56)**	0.12(3)	0.84(2)
WMaxcut/Random/	40	MiniMaxSAT - **5.43(40)**	-(0)	-(0)
WMaxcut/Spinglass/	5	MiniMaxSAT - **27.6(4)**	-(0)	-(0)
Weighted Partial MaxSAT Category				
Crafted				
Auctions/Auc_paths/	88	IncWMaxsatz - **8.4(88)**	-(0)	497(15)
Auctions/Auc_regions/	88	MiniMaxSAT - **1.7(84)**	-(0)	166(76)
Auctions/Auc_Sched/	84	MiniMaxSAT - **46(84)**	-(0)	317(49)
Random-net/	350	Clone - **72(236)**	194(91)	331(13)
Pseudo-factor/	186	IncWMaxsatz - **0.07(186)**	16(124)	3.3(186)
Pseudo- miplib/	16	SAT4J - 13(6)	0.29(3)	13(6)
QCP/	25	SAT4J - 6.14(25)	**0.27(25)**	6.14(25)
WCSP/Planning/	71	SAT4J - **6.55(71)**	0.9(46)	**6.55(71)**
WCSP/Spot5/Dir/	21	Clone - 87.6(6)	2.31(4)	76(3)
WCSP/Spot5/Log/	21	Clone - **15(6)**	0.52(5)	63.8(3)
Industrial				
Protein_ins	12	MiniMaxSAT - **482(8)**	42(1)	6.05(1)

interface to access to the unsatisfiable core when the formula is UNSAT. In order to encode the cardinality constraints, for WPM1, we use the regular encoding presented in [1], and for PM2 we use the encoding based on sequential counters presented in [18].

In SAT4J [4], for each clause c_i in the original problem, a new blocking variable b_i is added. Then a SAT solver is called to solve the new formula, and each time a model is found, a cardinality constraint is added to the formula that states that the sum of blocking variables has to be less than the number of blocking variables satisfied in the previous iteration. Once the SAT solver gives an UNSAT answer, the latest model is an optimal solution.

Msu1.2 [14,15] is another implementation of the Fu&Malik algorithm. Msu4.0 [16] is a more sophisticated approach, which alternates iterations to discover new cores with iterations to reduce the number of blocking variables that need to be set to true in each core.

Table 1 and Table 2 show the results of our experimental investigation. We set as timeout 1200 seconds. We report the number of solved instances (within parenthesis), and the mean time of the solved instances for each solver. The rules at the MaxSAT08 evaluation [2] establish that the winner is the solver

which solves more instances and ties are broken by selecting the solver with the minimum mean time. In bold we present the results of the winners.

We are interested in answering two questions: how our solvers would have performed at the MaxSAT08 evaluation [2], and how they compare to the current solvers based on satisfiability testing.

As we can see for the unweighted category, the solvers based on satisfiability testing perform well at the industrial category being our solver WPM1 the best performing one. For the crafted instances, the solvers based on satisfiability testing are not competitive, however our solver PM2 is the best among them.

For the partial category and the crafted instances, the solvers based on satisfiability testing are again not competitive, except for SAT4J [4] at one benchmark. However, for the industrial instances, they win 7 out of 9 benchmarks, in particular, WPM1 wins at 2, PM2 at 3 and msu4.0 [16] at 2.

For the weighted category, WPM1 and SAT4J [4] just show a good behavior on one set of instances, the Ramsey set. However, looking more closely to the set, many of the instances are actually satisfiable. Unfortunately, there are not available industrial instances for this category.

For the weighted partial category and crafted instances, WPM1 and SAT4J [4] are just able to win at 2 out 9 benchmarks. Again, unfortunately, there is only one set of industrial instances, and MinimaxSAT is the only solver able to solve 8 instances while the rest of the solvers submitted to the MaxSAT08 evaluation [2] and the one presented in this paper are not able to solve more than 2 instances.

As a whole, we can say that there is not a clear winner in all the categories, and so far, all the approaches can have some potential. Respect to the solvers we have presented in this paper, we have shown that our implementations show a good performance on the industrial instances for the unweighted and partial categories. That should be a promising base point for the weighted versions. Although we can not make such a claim yet, since there are not enough industrial instances at these categories in order to test our solvers, we think this research avenue is worth further investigation.

Acknowledgements

We especially thank Armin Biere for the insightful discussions about his SAT solver picosat. We would also like to thank the organizers of the MaxSAT evaluations.

References

1. Ansótegui, C., Manyà, F.: Mapping problems with finite-domain variables to problems with boolean variables. In: Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 1–15. Springer, Heidelberg (2005)
2. Argelich, J., Li, C.M., Manyà, F., Planes, J.: The first and second Max-SAT evaluations. Journal on Satisfiability 4, 251–278 (2008)

3. Argelich, J., Manyà, F.: Partial Max-SAT solvers with clause learning. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 28–40. Springer, Heidelberg (2007)
4. Berre, D.L.: Sat4jmaxsat, http://www.sat4j.org
5. Biere, A.: PicoSAT essentials. Journal on Satisfiability 4, 75–97 (2008)
6. Darras, S., Dequen, G., Devendeville, L., Li, C.M.: On inconsistent clause-subsets for Max-SAT solving. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 225–240. Springer, Heidelberg (2007)
7. Fu, Z.: Extending the Power of Boolean Satisfiability: Techniques and Applications. PhD thesis, Princeton University, Princeton (2007)
8. Fu, Z., Malik, S.: On solving the partial max-sat problem. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 252–265. Springer, Heidelberg (2006)
9. Heras, F., Larrosa, J.: New inference rules for efficient Max-SAT solving. In: Proc. the 21th National Conference on Artificial Intelligence (AAAI 2006) (2006)
10. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSat: A new weighted Max-SAT solver. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 41–55. Springer, Heidelberg (2007)
11. Li, C.M., Manyà, F., Planes, J.: New inference rules for Max-SAT. J. Artif. Intell. Res. (JAIR) 30, 321–359 (2007)
12. Lin, H., Su, K.: Exploiting inference rules to compute lower bounds for Max-SAT solving. In: Proc. of the 20th Int. Joint Conf. on Artificial Intelligence (IJCAI 2007), pp. 2334–2339 (2007)
13. Lin, H., Su, K., Li, C.M.: Within-problem learning for efficient lower bound computation in Max-SAT solving. In: Proc. the 23th National Conference on Artificial Intelligence (AAAI 2008), pp. 351–356 (2008)
14. Marques-Silva, J., Manquinho, V.M.: Towards more effective unsatisfiability-based maximum satisfiability algorithms. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 225–230. Springer, Heidelberg (2008)
15. Marques-Silva, J., Planes, J.: On using unsatisfiability for solving maximum satisfiability. CoRR, abs/0712.1097 (2007)
16. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Proc. of the Conf. on Design, Automation and Test in Europe (DATE 2008), pp. 408–413 (2008)
17. Pipatsrisawat, K., Darwiche, A.: Clone: Solving weighted Max-SAT in a reduced search space. In: Australian Conference on Artificial Intelligence, pp. 223–233 (2007)
18. Sinz, C.: Towards an optimal CNF encoding of boolean cardinality constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)

Nonlinear Pseudo-Boolean Optimization: Relaxation or Propagation?

Timo Berthold[1,⋆], Stefan Heinz[1,⋆], and Marc E. Pfetsch[2]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
{berthold,heinz}@zib.de
[2] Technische Universität Braunschweig, Institut für Mathematische Optimierung, Pockelsstraße 14, 38106 Braunschweig, Germany
m.pfetsch@tu-bs.de

Abstract. Pseudo-Boolean problems lie on the border between satisfiability problems, constraint programming, and integer programming. In particular, nonlinear constraints in pseudo-Boolean optimization can be handled by methods arising in these different fields: One can either linearize them and work on a linear programming relaxation or one can treat them directly by propagation. In this paper, we investigate the individual strengths of these approaches and compare their computational performance. Furthermore, we integrate these techniques into a branch-and-cut-and-propagate framework, resulting in an efficient nonlinear pseudo-Boolean solver.

1 Introduction

Pseudo-Boolean (PB) optimization extends the satisfiability (SAT) problem by allowing integer coefficients in the constraints, multiplication of variables, and an objective function. As in SAT, variables take 0/1 (false/true) values.

There are several, fundamentally different, ways to attack the solution of PB-problems. One way is to apply a transformation to a SAT problem. This approach is used, for instance, in the solver MiniSat+ [10]. Another way is to handle PB-constraints directly in the solver, see, e.g., SAT4JPseudo [8], PBS [6], and pueblo [18]. Other solvers use a constraint programming approach, e.g., absconPseudo [13]. Pseudo-Boolean problems can also be formulated as an 0/1 integer program (IP), in which the nonlinear constraints are linearized. For instance glpPB uses this idea and applies the IP-solver glpk [12]. The solver bsolo [14] combines SAT-solving techniques with IP-methodologies to solve linear pseudo-Boolean problems, if the bounds from the linear programming (LP) relaxation are promising. The performance of the IP-solver Cplex for linear pseudo-Boolean problems was investigated in [5]. A variety of PB-solvers have been compared during the Pseudo-Boolean Evaluations [15,16].

In this paper, we approach nonlinear PB-problems via *constraint integer programming* (CIP). CIP is a combination of IP, SAT, and constraint programming

⋆ Supported by the DFG Research Center Matheon *Mathematics for key technologies* in Berlin.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 441–446, 2009.

(CP) methodologies. CIP was introduced by Achterberg [1,2] and implemented in the framework SCIP. The basic idea is to apply a branch-and-cut-and-propagate method. Hence, one performs a branch-and-bound algorithm to decompose the problem into subproblems (as in SAT, CP, and IP-solvers). One solves a linear relaxation, which is strengthened by additional inequalities/cutting planes if possible (as in IP-solvers). One uses propagation techniques (similar to CP-solvers) in the nodes of the search tree. Moreover, one applies conflict analysis and restarts (similar to SAT-solvers). Detailed descriptions of the CIP-paradigm and the algorithmic design of SCIP can be found in [1,2,3].

The main goal when applying CIP to PB problems is to use the IP-machinery with LP-relaxations, cutting planes, elaborated branching rules, etc., and directly propagating the (nonlinear) multiplications. As far as we know, all of the PB-solvers discussed above *either* handle nonlinearities directly *or* add a complete linearization to the problem formulation. We compare both ideas and introduce a hybrid approach which only partially linearizes the nonlinear part of the problem. It turns out that the combination of both, propagation and (partial) linear relaxation, performs better than applying only one of these.

2 Problem Definition

For a Boolean variable $x \in \{0,1\}$, a *literal* ℓ is either the original variable x or its negation $\overline{x} := 1 - x$. A *(nonlinear) pseudo-Boolean problem* with Boolean variables $x_1, \ldots, x_n$ ($n \in \mathbb{N}$) is an optimization problem of the following form:

$$\begin{aligned} \min \quad & \sum_{j=1}^{t_0} c_j \cdot \prod_{\ell \in I_{0j}} \ell && (1)\\ & \sum_{j=1}^{t_i} a_{ij} \cdot \prod_{\ell \in I_{ij}} \ell \ \geq b_i && \text{for } i = 1, \ldots, m \\ & \boldsymbol{x} \in \{0,1\}^n. \end{aligned}$$

Here, $m \in \mathbb{N}$ defines the number of constraints, I_{ij} is a subset of literals for $i = 0, \ldots, m$ and $j = 1, \ldots, t_i$, where $t_i \in \mathbb{N}$ is the number of summands in constraint i. All coefficients a_{ij}, b_i, c_j are required to be integral. Let

$$\mathcal{I} := \{(i,j) \,:\, i \in \{0, \ldots, m\},\ j \in \{1, \ldots, t_i\}\}.$$

The above formulation is quite general: one can easily incorporate maximization, "$\leq$" constraints, equations, and pure satisfiability problems. If $|I_{ij}| \leq 1$ for all $(i,j) \in \mathcal{I}$, the objective function and the constraints are linear expressions in the variables. We call such instances *linear pseudo-Boolean problems*. If the objective function equals zero (or any other constant), we have *satisfiability* problems, otherwise *optimization* problems.

SAT problems are special cases of PB problems with $b_i = 1$, for $i = 1, \ldots, m$, $c_j = 0$, for $j = 1, \ldots, t_0$, and $a_{ij} \in \{0,1\}$ for $i = 1, \ldots, m$, $j = 1, \ldots, t_i$.

3 Handling of Nonlinearities

Linear Relaxations. To deal with the nonlinear constraints, we transform Problem (1) as follows. For each $(i,j) \in \mathcal{I}$ with $|I_{ij}| > 1$, we introduce a new Boolean variable $z_{ij} = \prod_{\ell \in I_{ij}} \ell$. The product can also be seen as an AND-expression $z_{ij} = \bigwedge_{\ell \in I_{ij}} \ell$, for which we apply the following linearization:

$$z_{ij} - \ell \leq 0 \qquad \text{for } \ell \in I_{ij} \tag{2}$$

$$\sum_{\ell \in I_{ij}} \ell - z_{ij} \leq |I_{ij}| - 1. \tag{3}$$

After replacing the AND-expressions by z_{ij}, the resulting constraint is linear in z_{ij}. This linearization has the following nice feature – we omit the proof.

Lemma. *The polyhedron defined by (2), (3), $z_{ij} \geq 0$, and $\ell \leq 1$ for all $\ell \in I_{ij}$ is integral, i.e., has only integral vertices.*

Note that the above linearization is different from

$$\sum_{\ell \in I_{ij}} \ell - |I_{ij}|\, z_{ij} \geq 0, \qquad \sum_{\ell \in I_{ij}} \ell - z_{ij} \leq |I_{ij}| - 1, \tag{4}$$

which is used in the Pseudo-Boolean Evaluation [16]. Both linearizations have the property that 0/1-solutions of the corresponding systems are solutions of their AND-expressions and conversely. However, while the above Lemma holds for the first linearization, the corresponding polyhedron of (4) is not integral, since $z_{ij} = \frac{1}{n}$, $\hat{\ell} = 1$ for some $\hat{\ell} \in I_{ij}$, $\ell = 0$ for all $\ell \in I_{ij} \setminus \hat{\ell}$ is a fractional vertex.

Linearization (4) has the advantage that it contains only two constraints, compared to $|I_{ij}|+1$ in (2) and (3). The larger size of the first linearization can be handled by a so-called *separation mechanism*, i.e., the necessary inequalities are generated on the fly, if they are violated. More precisely, one solves an LP which initially only consists of the objective function and the linear constraints in z_{ij} and neglect the AND-expressions (and integrality constraints). Inequalities (2) and (3) are added, only if they violate the optimal solution of this LP-relaxation. Then the resulting LP is solved and the process is iterated.

The advantages and disadvantages of the above linearizations have been widely discussed in the literature, see, for instance, Glover and Woolsey [11], Balas and Mazzola [7], and Adams and Sherali [4].

Constraint Programming. The CP approach applies a domain propagation algorithm at each subproblem of the branch-and-bound process in order to fix further variables. The propagation rules are as follows. If one of the operand variables $\ell \in I_{ij}$ is fixed to zero the resultant variable z_{ij} has to be zero, too. On the other hand, if all operand variables are set to one, the resultant variables must also be fixed to one, and vice versa. Finally, if the resultant variable z_{ij} is zero and all but one of the operand variables are one, the remaining operand variable can be fixed to zero.

The main advantage of this approach is that all these propagation rules can be applied very efficiently and therefore the computation time per node is very

small. The disadvantage is that one looses the global view of the LP-relaxation and its strong capability of pruning suboptimal parts of the tree.

Constraint Integer Programming. The hope of an integrated approach is that on the one hand the fixings derived by domain propagation reduce the size of the LP and therefore potentially the computational overhead. On the other hand, these fixings may even yield a stronger LP-bound which vice versa can lead to further variable fixings which can be propagated and so forth.

We study three different variants of integration. First, we apply the suggested separation mechanism simultaneously with the propagation algorithm. Second, we add the complete linearization and apply propagation. Third, we change the strategy dynamically depending on the problem's degree of nonlinearity.

4 Computational Results

In this section, we analyze how each of the approaches performs for the nonlinear test sets of the Pseudo-Boolean Evaluation 2007 [16].

All computations reported in the following were obtained using version 1.1.0.6 of SCIP [17] on Intel Xeon Core 2.66 GHz computers (in 64 bit mode) with 4 MB cache, running Linux, and 6 GB of main memory. We integrated CLP release version 1.9.0 as underlying LP-solver [9]. Thus, we only used noncommercial software, which is available in source code.

As in the PB evaluation, we set a time limit of 1800 seconds. We compared the performance of SCIP for six different settings, which only differ in the way they handle the AND-expressions. The setting "only relaxation" only applies the complete linearization of the AND-expressions before starting the search, "only separation" only uses separation, i.e., adding inequalities (2) and (3) when they are violated, and "only propagation" only performs propagation without using the linearization. The settings "relaxation/propagation" and "separation/propagation" linearize the AND-expressions in advance and on the fly, respectively. Additionally, they apply the described propagation algorithms, thereby combining CP and IP techniques. The setting "dynamic" incorporates the latter two: If the linearization of the AND-expressions consists of less than 10 000 linear constraints, "relaxation/propagation" will be used, otherwise, "separation/propagation" will be used. The motivation was to work on the complete linear description only if it is small and not likely to produce a huge computational overhead. We use the inequalities (2) and (3) as linearization. All remaining parameters of SCIP were set to their default values, hence we use primal heuristics such as the feasibility pump, general purpose cutting planes such as Gomory cuts, preprocessing strategies, and we use conflict analysis and restarts.

According to the PB evaluation, the instances are split into the following two groups, both with "small" integers, i.e., all coefficients are representable as 32 bit integers: OPT-SMALLINT-NLC (nonlinear PB optimization), SATUNSAT-SMALLINT-NLC (nonlinear PB satisfiability). For details we refer to [16].

We compare for how many instances optimality ("opt") or at least satisfiability ("sat") could be proven, the number of instances for which no result was obtained ("unkn"), total time and number of branch-and-bound nodes over all

Table 1. Results for the 405 OPT-SMALLINT-NLC instances

Setting	opt	sat	unkn	Nodes total(k)	Nodes geom[1]	Time in [s] total(k)	Time in [s] geom[1]
only propagation	269	321	84	152027	13477	235.3	62.6
only separation	225	276	129	67313	5583	359.3	202.5
only relaxation	236	326	79	90639	4184	340.5	194.0
separation/propagation	288	341	64	12219	1267	225.2	61.5
relaxation/propagation	284	372	33	5105	846	226.3	59.6
dynamic	291	342	63	11009	1219	223.5	64.3
MINISAT+	279	397	8	–	–	234.0	46.2

instances in the test set and the shifted geometric means[1] ("geom") over these two performance measures. For the satisfiability test set, we compare the number of instances for which an answer could be found ("solved"), which we subdivide into feasible ("sat") and infeasible ("unsat") instances, the time and the number of branch-and-bound nodes in total and in shifted geometric mean as before.

The results of Tables 1 and 2 show that the combined approaches are superior to the ones which use only one algorithm. For the optimization instances, each of them solves more instances to optimality and finds more feasible solutions than each of the "only" settings. The same holds for the number of solved instances in the satisfiability test set. Furthermore, the combined approaches usually need less branch-and-bound nodes and less overall running time.

As one would expect, the setting "relaxation/propagation", the method with the highest computational effort, needs the fewest branch-and-bound nodes for both test sets, but spends the most time per node. In contrast, "only propagation" requires little time per node, but needs the most branch-and-bound nodes.

The "dynamic" setting enables to solve most of the optimization problems and only one instance less than the best setting for the satisfiability instances. The setting "relaxation/propagation" is the best performing for the satisfiability instances and, moreover, the best in finding feasible solutions for the optimization problems. This can be explained by the fact that the primal heuristics work best, if there is a full linear description present.

We conclude that combining LP-relaxation and domain propagation techniques help to solve nonlinear pseudo-Boolean problems. Furthermore, for proving optimality, it is recommendable to only use a partial linearization for instances with a large nonlinear part. We also performed all experiments using the linearization (4). The results are similar, but slightly worse.

For comparison, we ran MINISAT+, the best solver for nonlinear PB problems in the PB evaluation 2007, on the same computational environment. For the results in Tables 1 and 2 we used linearization (2) and (3), while with linearization (4) MINISAT+ solved two instances less of the optimization test set and performed slightly worse in both cases.

[1] The shifted geometric mean of values $t_1, \ldots, t_n$ is defined as $\left(\prod(t_i + s)\right)^{1/n} - s$ with shift s. We use a shift $s = 10$ for time and $s = 100$ for nodes in order to decrease the strong influence of the very easy instances in the mean values.

Table 2. Results for the 100 SatUnsat-Smallint-Nlc instances

Setting	solved	sat	unsat	unkn	Nodes total(k)	Nodes geom[1]	Time in [s] total(k)	Time in [s] geom[1]
only propagation	60	50	10	40	41085	6883	72.5	86.5
only separation	70	50	20	30	671	281	55.0	57.3
only relaxation	71	51	20	29	446	161	58.4	69.6
separation/propagation	72	52	20	28	883	284	51.7	53.3
relaxation/propagation	73	53	20	27	489	154	55.7	66.4
dynamic	72	52	20	28	835	279	51.8	55.6
MiniSat+	65	50	15	35	–	–	63.1	56.4

References

1. Achterberg, T.: Constraint Integer Programming. PhD thesis, TU Berlin (2007)
2. Achterberg, T.: SCIP: solving constraint integer programs. Mathematical Programming Computation, 1 (2008)
3. Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: A new approach to integrate CP and MIP. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 6–20. Springer, Heidelberg (2008)
4. Adams, W.P., Sherali, H.D.: Linearization strategies for a class of zero-one mixed integer programming problems. Oper. Res. 38(2), 217–226 (1990)
5. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Generic ILP versus specialized 0-1 ILP: an update. In: Pileggi, L.T., Kuehlmann, A. (eds.) Proc. of the 2002 IEEE/ACM International Conference on Computer-aided Design, pp. 450–457. ACM, New York (2002)
6. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: PBS: A backtrack-search pseudo-boolean solver and optimizer. In: Proc. Fifth International Symposium on Theory and Applications of Satisfiability Testing (SAT 2002), pp. 346–353 (2002)
7. Balas, E., Mazzola, J.B.: Nonlinear 0-1 programming: I. Linearization techniques. Math. Prog. 30(1), 1–21 (1984)
8. Berre, D.L.: Sat4j, http://www.sat4j.org/
9. Clp. COIN-OR LP-solver, http://www.coin-or.org/projects/Clp.xml
10. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. J. Satisf. Boolean Model. Comput. 2, 1–26 (2006)
11. Glover, F., Woolsey, E.: Converting the 0-1 polynomial programming problem to a 0-1 linear program. Oper. Res. 22(1), 180–182 (1974)
12. GLPK. GNU linear programming kit, http://www.gnu.org/software/glpk/
13. Hemery, F., Lecoutre, C.: AbsconPseudo 2006 (2006), http://www.cril.univ-artois.fr/PB06/papers/abscon2006V2.pdf
14. Manquinho, V.M., Marques-Silva, J.: On using cutting planes in pseudo-Boolean optimization. J. Satisf. Boolean Model. Comput. 2, 209–219 (2006)
15. Manquinho, V.M., Roussel, O.: The first evaluation of pseudo-Boolean solvers (PB 2005). J. Satisf. Boolean Model. Comput. 2, 103–143 (2006)
16. Manquinho, V.M., Roussel, O.: Pseudo-Boolean evaluation 2007 (2007), http://www.cril.univ-artois.fr/PB07/
17. SCIP. Solving Constraint Integer Programs, http://scip.zib.de/
18. Sheini, H.M., Sakallah, K.A.: Pueblo: A hybrid pseudo-boolean SAT solver. J. Satisf. Boolean Model. Comput. 2, 165–189 (2006)

Relaxed DPLL Search for MaxSAT*

Lukas Kroc, Ashish Sabharwal, and Bart Selman

Department of Computer Science
Cornell University, Ithaca NY 14853-7501, U.S.A.
{kroc,sabhar,selman}@cs.cornell.edu

Abstract. We propose a new incomplete algorithm for the Maximum Satisfiability (MaxSAT) problem on unweighted Boolean formulas, focused specifically on instances for which proving unsatisfiability is already computationally difficult. For such instances, our approach is often able to identify a small number of what we call "bottleneck" constraints, in time comparable to the time it takes to prove unsatisfiability. These bottleneck constraints can have useful semantic content. Our algorithm uses a relaxation of the standard backtrack search for satisfiability testing (SAT) as a guiding heuristic, followed by a low-noise local search when needed. This allows us to heuristically exploit the power of unit propagation and clause learning. On a test suite consisting of all unsatisfiable industrial instances from SAT Race 2008, our solver, `RelaxedMinisat`, is the only (MaxSAT) solver capable of identifying a *single* bottleneck constraint in all but one instance.

1 Introduction

In recent years, we have seen tremendous progress in the area of Boolean Satisfiability (SAT) solvers. Current solvers can handle instances with over a million variables and millions of clauses. These advances have led to an ever growing range of applications, particularly in hardware and software verification, and planning. In fact, the technology has matured from being a largely academic endeavor to an area of research with strong academic and industrial participation. The current best SAT solvers for handling "structured" instances are based on Davis-Putnam-Logemann-Loveland (DPLL) style complete search.

Determining whether a Boolean formula is satisfiable or not is a special case of the maximum satisfiability (MaxSAT) problem, where the goal is to find an assignment that satisfies as many clauses as possible. Even though MaxSAT is a natural generalization of SAT, and thus closely related, progress has been much slower on the MaxSAT problem. There is a good explanation as to why this is the case. Two of the key components behind the rapid progress for DPLL-based SAT solvers are highly effective unit propagation and clause learning.

* This research was supported by IISI, Cornell University (AFOSR grant FA9550-04-1-0151), NSF Expeditions in Computing award for Computational Sustainability (Grant 0832782) and NSF IIS award (Grant 0514429). Part of this work was done while the second author was visiting McGill University.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 447–452, 2009.

Both techniques in a sense focus on avoiding local inconsistencies: when a unit clause occurs in a formula, one should immediately assign the appropriate truth value to the variable so that it satisfies the clause, and when a branch reaches a contradiction, a no-good clause can be derived which captures the cause of the local inconsistency. In a MaxSAT setting, these strategies can be quite counterproductive and inaccurate. For example, for an unsatisfiable instance, the best assignment, i.e., one satisfying the most clauses, may be the one that violates several unit clauses. Also, when a contradiction is reached, the best solution may be to violate one of the clauses that led to the contradiction rather than adding a no-good clause which effectively steers the search away from the contradiction. So, neither unit propagation nor clause learning appear directly suitable for a MaxSAT solver.[1] Unfortunately, taking such mechanisms out of DPLL search dramatically reduces its effectiveness. This is confirmed when one considers the performance of exact solvers for MaxSAT that, in effect, employ a branch-and-bound search but do not have unit propagation or clause learning incorporated. The MaxSAT instances that can be solved by exact solvers in practice are generally much smaller than instances that can be handled by SAT solvers [cf. 6].

The question we ask in this work is the following: *can traditional SAT solver techniques like unit propagation and clauses learning be directly used as effective heuristics for an incomplete MaxSAT algorithm?* Our results demonstrate that the answer is clearly affirmative, as long as the instances are not too easy to prove unsatisfiable. Specifically, we consider a relaxation of the standard DPLL-based solver where it is allowed to essentially *ignore* the first ℓ conflicts. We show that this strategy is able to solve challenging industrial MaxSAT instances that are hard to prove unsatisfiable, specifically *all* unsatisfiable instances from SAT Race 2008 [10], better than currently available exact as well as approximate alternative techniques. In a few of these instances where the candidate MaxSAT solution found by this approach is sub-optimal, we show that performing a low-noise (and hence very greedy and fast) local search initiated at this candidate solution is often results in an optimal MaxSAT solution within seconds.[2]

Bottleneck Constraints: Interestingly, our approach revealed that the optimal solutions for all but one of the unsatisfiable industrial instances from SAT Race 2008 have only *one* violated clause. Note that, in contrast, all other MaxSAT solvers incorrectly suggest that these very instances have hundreds, if not thousands, of violated clauses in the best MaxSAT solutions. This one clause can be thought of as a *bottleneck constraint* whose semantics can sometimes be used to guide the problem designer towards appropriate additional resources that may be acquired to turn the instance into a feasible one. Of course, it is unclear that a bottleneck constraint always provides meaningful semantic information about additional resources that are realistic to acquire; the violated constraint could, in principle, be a "frame axiom" or "consistency constraint" enforcing that the encoding is meaningful.

[1] Unit propagation is used *indirectly* in many exact MaxSAT solvers to generate good lower bounds [cf. 6, 8], and so are resolution-based inference mechanisms [6]. We compare our results against these approaches as well as local search methods.

[2] Performing local search also works for instances that are too easy to prove unsatisfiable and thus not ideal for us; here our method falls back to pure local search.

To investigate this further, we conducted experiments on AI planning instances for which we have full knowledge of the variable and clause semantics. Specifically, we considered the TPP (Traveling Purchase Problem) domain from the IPC-5 Competition [3], giving the planner one fewer time step than what it needs to solve the instance. We translated instances of this infeasible problem into unsatisfiable SAT formulas following the SatPlan framework [4]. Our solver, `RelaxedMinisat`, was able to identify a single violated clause in instances thus generated. With different random seeds, we obtained a number of different "explanations" of the unsatisfiability of each instance, in the form of a bottleneck constraint. As an example, in an instance involving 10 goods to be purchased in various quantities and transported, 1 storage depot, 3 markets, and 3 trucks, a violated bottleneck constraint had the following precondition-style semantics: "in order to drive truck-3 from depot-1 to market-2 at time-step-7, truck-3 must be at depot-1 at the end of time-step-6." One thing this bottleneck constraint suggests is that there is a plan that "almost" works if the post condition achieved by the action under consideration is somehow made to hold, i.e., if we could somehow make a truck available at market-2 at the end of time-step-7. Indeed, if we add a fourth truck as a new resource at market-2 as part of the initial conditions, the instance becomes satisfiable in the given time steps.

This suggests that the small number of bottleneck constraints identified by our approach can provide useful semantic information about potential additional resources that can make the problem feasible. This information is complementary to that provided by, for example, "minimal unsatisfiable cores" and related concepts [cf. 8]. For the specific example discussed above, the minimal unsat core returned by the `zChaff` solver involves 522 clauses. For a more detailed discussion of this motivation, we refer the reader to our recent related work [5].

2 Using Relaxed DPLL as a Heuristic for MaxSAT

Due to lack of space we assume familiarity with Boolean formulas in conjunctive normal form (CNF), the satisfiability testing problem (SAT), the maximum satisfiability problem (MaxSAT), the standard DPLL style systematic backtrack search for SAT with conflict clause learning, and basic local search. The reader may want to refer to the Handbook of Satisfiability [1] for a review.

The idea behind our solver, `RelaxedMinisat`, is relatively simple: use a state-of-the-art DPLL solver such as `Minisat` [2] but relax it to "ignore" a fixed number ℓ of conflicts on each search branch, and quit once a truth assignment violating at most ℓ clauses is found. If necessary, run a low-noise local search initiated at the partial assignment found by the DPLL solver.

We chose `Minisat` [2] as the DPLL solver to build on. To implement the one extra feature we need—allowing up to ℓ conflicts on a search branch—we slightly modify the routine performing unit propagation. When a conflict is detected on a search branch b and it is amongst the first ℓ conflicts along b, the clause causing the conflict is silently ignored until the solver later backtracks the closest branching decision made before getting to this point. All other functionality of the solver is left intact, including clause learning. If ℓ conflicts are reached on

the branch b, conflict directed backtracking is performed as usual. It is not hard to see that the conflict clause C learned at this point has the following property: *if any truth assignment σ satisfies all clauses except the ℓ conflict generating clauses seen on branch b, then σ also satisfies C.* Adding C as a learned clause therefore preserves the soundness of the technique. (But C could, in principle, rule out other potential solutions that violate a different set of ℓ or fewer clauses; adding C therefore does not preserve completeness.) If a solution is found before reaching ℓ conflicts, this solution is reported as a candidate MaxSAT solution; this provides an upper bound for the optimal MaxSAT solution. Alternatively, `RelaxedMinisat` can return the "UNSAT" status, in which case we increase the parameter ℓ to attempt to find some other truth assignment. (The "UNSAT" status of `RelaxedMinisat` does *not* mean that there is no assignment violating at most ℓ clauses.) Using binary search, one can find the smallest value of ℓ for which `RelaxedMinisat` does report an assignment; for nearly satisfiable instances such as the ones with very few bottleneck constraints that we focus on, this is very quick as ℓ is small. Experiments suggest that *rapid restarting* improves the performance of `RelaxedMinisat`. We restart after every 100 conflicts.

For some of the harder formulas, the candidate solution found by `RelaxedMinisat` was not clearly optimal (i.e., had more than one violated clause). In this case, we ran the local search solver `Walksat` [9] (without any modification) with the search initiated at this candidate truth assignment, looking for a better solution. Empirically, we observed that rapid restarts are again beneficial, along with very low noise to make the local search extremely greedy and focused.

3 Experimental Results

We conducted experiments on all 52 unsatisfiable formulas from the SAT-Race 2008 suite [10], which are all non-trivial to prove unsatisfiable and, as we will see, often beyond the reach of existing MaxSAT solvers. The solvers used in the comparison were from four families: exact MaxSAT solvers `maxsatz` [6] and `msuf` [8]; local search SAT solvers `saps` [11] and `adaptg2wsat+p` [7]; our previous hybrid approach `MiniWalk` [5] which tries to combine the power of DPLL and local search methods; and `RelaxedMinisat`. We used a cluster of 3.8 GHz Intel Xeon computers running Linux 2.6.9-22.ELsmp with a time limit of 1 hour (see two exceptions discussed below) and a memory limit to 2 GB.

The exact MaxSAT solvers selected were those that performed exceptionally well in MaxSAT Evaluation 2007, especially on industrial instances. The local search algorithms were selected as the best performing ones on our suite from a wide pool of choices offered by the `UBCSAT` solver [12]. Three runs for each problem and local search algorithm were performed with default parameters (or those used in the accompanying papers for the solvers, e.g., $\alpha = 1.05$ for `saps`), and the best run is reported.

For `RelaxedMinisat`, we first run the modified DPLL part allowing at most one conflict ($\ell = 1$), and increase this relaxation parameter when necessary. We

Table 1. Comparison of MaxSAT results for exact, local search, hybrid, and currently proposed methods. Timelimit: 1 hour (except for two instance which took two hours). If a sure optimum was achieved (i.e., one unsatisfied clause), the time is reported in parenthesis. The `RelaxedMinisat` column shows the final number of unsatisfied clauses reached, and in parenthesis the number of allowed conflicts ℓ, time for the DPLL part, and, if applicable, time for local search.

			Exact	Local Search		Hybrid	**Relaxed DPLL**
Instance	#vars	#cls	#unsat	best #unsat		best #uns	best #unsat
			`maxsatz` or `msuf`	`Adapt-` `g2wsat+p`	`SAPS`	`MiniWalk`	`RelaxedMinisat`
babic-dspam-vc1080	118K	375K	—	728	306	20	**1 (1,35s,–)**
babic-dspam-vc973	274K	908K	—	2112	1412	267	**1 (4,100s,44s)**
ibm-2002-22r-k60	209K	851K	—	198	409	10	**1 (3,115m,1s)**
ibm-2002-24r3-k100	148K	550K	—	205	221	2	**1 (1,7m,–)**
manol-pipe-f7nidw	310K	923K	—	810	797	7	**1 (1,3m,–)**
manol-pipe-f9b	183K	547K	—	756	600	177	**1 (1,8s,–)**
manol-pipe-g10nid	218K	646K	—	585	727	27	**1 (1,12s,–)**
manol-pipe-g8nidw	121K	358K	—	356	336	7	**1 (1,6s,–)**
post-c32s-col400-16	286K	840K	—	88	111	698	**1 (50,1m,8s)**
post-c32s-gcdm16-23	136K	404K	—	25	225	127	**1 (3,100m,1m)**
post-cbmc-aes-ele	277K	1601K	—	864	781	2008	**1 (1,14s,–)**
simon-s03-fifo8-400	260K	708K	—	89	289	13	**1 (1,11m,–)**
aloul-chnl11-13	286	1742	—	**4**	**4**	**4**	**4 (4,3s,×)**
anbul-dated-5-15-u	152K	687K	—	12	22	1 (15m)	**1 (1,8s,–)**
een-pico-prop05-75	77K	248K	—	2	47	**1 (4s)**	1 (1,2m,–)
fuhs-aprove-15	21K	74K	—	35	31	**1 (0s)**	**1 (1,0s,–)**
fuhs-aprove-16	52K	182K	—	437	246	1 (1s)	**1 (1,0s,–)**
goldb-heqc-dalumul	9426	60K	—	11	10	**1 (0s)**	1 (1,1s,–)
goldb-heqc-frg1mul	3230	21K	—	**1 (0s)**	**1 (0s)**	**1 (0s)**	**1 (1,0s,–)**
goldb-heqc-x1mul	8760	56K	—	**1 (0s)**	**1 (0s)**	**1 (0s)**	**1 (1,0s,–)**
hoons-vbmc-lucky7	8503	25K	—	**1 (0s)**	3	9	1 (7,27s,36s)
ibm-2002-25r-k10	61K	302K	—	111	95	1 (9s)	**1 (1,2s,–)**
ibm-2002-31_1r3-k30	44K	194K	—	78	101	**1 (2s)**	1 (1,6s,–)
ibm-2004-29-k25	17K	78K	—	14	12	1 (6m)	**1 (1,5s,–)**
manol-pipe-c10nid_i	253K	751K	—	678	695	1 (20m)	**1 (1,13s,–)**
manol-pipe-c10nidw	434K	1292K	—	1013	1363	**1 (16s)**	1 (1,37m,–)
manol-pipe-c6bidw_i	96K	284K	—	239	274	1 (24s)	**1 (1,3s,–)**
manol-pipe-c8nidw	269K	800K	—	697	742	**1 (7s)**	1 (1,6m,–)
manol-pipe-c9n_i	35K	104K	—	214	66	1 (3s)	**1 (1,0s,–)**
manol-pipe-g10bid_i	266K	792K	—	723	822	1 (103s)	**1 (1,18s,–)**
post-c32s-ss-8	54K	148K	—	1 (2s)	1 (8s)	**1 (0s)**	**1 (1,0s,–)**
post-cbmc-aes-d-r2	278K	1608K	—	834	734	1 (69s)	**1 (1,8s,–)**
post-cbmc-aes-ee-r2	268K	1576K	—	839	760	1 (37s)	**1 (1,12s,–)**
post-cbmc-aes-ee-r3	501K	2928K	—	1817	1822	1 (37m)	**1 (1,47s,–)**
schup-l2s-abp4-1-k31	15K	48K	—	7	16	**1 (0s)**	1 (1,2s,–)
schup-l2s-bc56s-1-k391	561K	1779K	—	5153	26312	**1 (168s)**	1 (1,11m,–)
simon-s02-f2clk-50	35K	101K	—	1 (110s)	32	**1 (12s)**	1 (1,17s,–)
velev-vliw-uns-2.0-iq1	25K	261K	—	1 (40m)	4	**1 (0s)**	**1 (1,0s,–)**
velev-vliw-uns-2.0-iq2	44K	542K	—	2	2	**1 (1s)**	**1 (1,1s,–)**
velev-vliw-uns-2.0-uq5	152K	2466K	—	40	11	1 (18s)	**1 (1,4s,–)**
velev-vliw-uns-4.0-9-i1	96K	1814K	—	12	10	1 (23s)	**1 (1,4s,–)**
velev-vliw-uns-4.0-9	154K	3231K	—	2	3	1 (10s)	**1 (1,3s,–)**
babic-dspam-vc949	113K	360K	1 (315s)	797	216	250	**1 (2,10s,0s)**
cmu-bmc-barrel6	2306	8931	1 (19m)	**1 (0s)**	**1 (0s)**	**1 (0s)**	**1 (1,0s,–)**
cmu-bmc-longmult13	6565	20K	1 (171s)	5	12	1 (1s)	**1 (1,0s,–)**
cmu-bmc-longmult15	7807	24K	1 (137s)	6	4	1 (5s)	**1 (1,0s,–)**
een-pico-prop00-75	94K	324K	1 (4m)	23	108	276	**1 (2,2m,1s)**
goldb-heqc-alu4mul	4736	30K	1 (14m)	1 (105s)	1 (47m)	1 (1s)	**1 (1,0s,–)**
jarvi-eq-atree-9	892	3006	1 (158s)	**1 (0s)**	**1 (0s)**	**1 (0s)**	**1 (1,0s,–)**
marijn-philips	3641	4456	1 (336)	**1 (0s)**	**1 (0s)**	**1 (0s)**	**1 (1,0s,–)**
post-cbmc-aes-d-r1	41K	252K	1 (177s)	7	10	**1 (1s)**	**1 (1,1s,–)**
velev-engi-uns-1.0-4nd	7000	68K	1 (76s)	1 (3s)	2	**1 (0s)**	**1 (1,0s,–)**

report in Table 1 the final number of unsatisfied clauses reached, followed by, in parentheses, the relaxation parameter, the time taken by the DPLL part, and the additional time taken to run the local search part afterwards (if applicable). In fact, in only 8 instances the DPLL part alone did not find the sure optimum solution; for these instances we increase ℓ (not necessarily by one) to the number reported in the table so that a candidate truth assignment is found. The table reports the time for the last run of `RelaxedMinisat`, with the appropriate value of ℓ (often 1). On only two instances did `RelaxedMinisat` require more than one hour (ibm-2002-22r-k60 and post-c32s-gcdm16-23). For local search, we use the default quick restarts of `Walksat` every 100,000 flips. (However, for post-c32s-col400-16 and post-c32s-gcdm16-23 instances, this was increased to 1 million.) In all cases the noise parameter for `Walksat` was set to a low value of 5%.

Table 1 shows that `RelaxedMinisat` is the only solver able to solve 51 of out 52 instances to optimality. (For aloul-chnl11-13, we know that the optimum is not 1.) Moreover, 39 of these instances, i.e. 76%, were solved in under one minute. In contrast, the best local search approaches often could not find a solution violating less than a few hundred or thousand clauses in one hour of computation time. The hybrid method, `MiniWalk`, showed better performance but was still unable to solve to optimality 16 instances out of 52 (i.e., 30% of the instances). These results demonstrate that our relaxed DPLL solver is able to efficiently identify bottleneck constraints in challenging unsatisfiable problem instances.

References

[1] Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in AI and Applications, vol. 185. IOS Press, Amsterdam (2009)
[2] Eén, N., Sörensson, N.: MiniSat: A SAT solver with conflict-clause minimization. In: 8th SAT, St. Andrews, UK (June 2005)
[3] Gerevini, A., Dimopoulos, Y., Haslum, P., Saetti, A. (Organizers): IPC-5 international planning competition (June 2006), http://zeus.ing.unibs.it/ipc-5
[4] Kautz, H.A., Selman, B.: BLACKBOX: A new approach to the application of theorem proving to problem solving. In: Workshop on Planning as Combinatorial Search, held in conjunction with AIPS 1998, Pittsburgh, PA (1998)
[5] Kroc, L., Sabharwal, A., Gomes, C.P., Selman, B.: Integrating systematic and local search paradigms: A new strategy for MaxSAT. In: 21st IJCAI (July 2009)
[6] Li, C.M., Manyà, F., Planes, J.: New inference rules for Max-SAT. JAIR 30 (2007)
[7] Li, C.M., Wei, W., Zhang, H.: Combining adaptive noise and look-ahead in local search for SAT. In: 10th SAT, Lisbon, Portugal, May 2007, pp. 121–133 (2007)
[8] Marques-Silva, J.P., Manquinho, V.M.: Towards more effective unsatisfiability-based maximum satisfiability algorithms. In: 11th SAT, May 2008, pp. 225–230 (2008)
[9] Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: The Second DIMACS Implementation Challenge, pp. 521–532 (1996)
[10] Sinz, C. (Organizer).: SAT-race 2008 (May 2008)
[11] Tompkins, D.A.D., Hoos, H.H.: Scaling and probabilistic smoothing: Dynamic local search for unweighted MAX-SAT. In: 16th Canadian AI, pp. 145–159 (2003)
[12] Tompkins, D.A.D., Hoos, H.H.: UBCSAT: An impl. and experim. env. for SLS algorithms for SAT and MAX-SAT. In: 7th SAT, Vancouver, BC (May 2004)

Branch and Bound for Boolean Optimization and the Generation of Optimality Certificates

Javier Larrosa, Robert Nieuwenhuis,
Albert Oliveras, and Enric Rodríguez-Carbonell*

Abstract. We consider optimization problems of the form $(S, cost)$, where S is a clause set over Boolean variables $x_1 \dots x_n$, with an arbitrary cost function $cost\colon \mathbb{B}^n \to \mathbb{R}$, and the aim is to find a model A of S such that $cost(A)$ is minimized.

Here we study the generation of *proofs of optimality* in the context of branch-and-bound procedures for such problems. For this purpose we introduce $\mathtt{DPLL}_{\mathtt{BB}}$, an abstract DPLL-based branch and bound algorithm that can model optimization concepts such as cost-based propagation and cost-based backjumping.

Most, if not all, SAT-related optimization problems are in the scope of $\mathtt{DPLL}_{\mathtt{BB}}$. Since many of the existing approaches for solving these problems can be seen as instances, $\mathtt{DPLL}_{\mathtt{BB}}$ allows one to formally reason about them in a simple way and exploit the enhancements of $\mathtt{DPLL}_{\mathtt{BB}}$ given here, in particular its uniform method for generating independently verifiable optimality proofs.

1 Introduction

An important issue on algorithms for Boolean satisfiability is their ability to provide proofs of unsatisfiability, so that also negative answers can be verified with a trusted independent proof checker. Many current SAT solvers provide this feature typically by writing (with little overhead) a trace file from which a resolution proof can be reconstructed and checked.

In this paper we address a related topic. We take a very general class of Boolean optimization problems and consider the problem of computing the best model of a CNF with respect to a cost function and, additionally, a proof of its optimality. The purpose of the paper is to provide a general solving framework that is faithful to state-of-the-art branch-and-bound solvers and where it is simple to reason about them and to generate optimality proofs. We show how branch-and-bound algorithms can provide proofs with little overhead, as in the SAT case. To the best of our knowledge, no existing solvers offer this feature.

The first contribution of the paper is an abstract DPLL-like branch-and-bound algorithm ($\mathtt{DPLL}_{\mathtt{BB}}$) that can deal with most, if not all, Boolean optimization problems considered in the literature. $\mathtt{DPLL}_{\mathtt{BB}}$ is based on standard abstract DPLL rules and includes features such as propagation, backjumping, learning

* All authors from Technical Univ. of Catalonia, Barcelona, and partially supported by Spanish Min. of Science and Innovation through the projects TIN2006-15387-C03-0 and TIN2007-68093-C02-01 (LogicTools-2).

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 453–466, 2009.

or restarts. The essential difference between classical DPLL and its branch-and-bound counterpart is that the rules are extended from the usual SAT context to the optimization context by taking into account the cost function to obtain entailed information. Thus, $\texttt{DPLL}_{\texttt{BB}}$ can model concepts such as, e.g., cost-based propagation and cost-based backjumping. To exploit the cost function in the search with these techniques, $\texttt{DPLL}_{\texttt{BB}}$ assumes the existence of a lower bounding procedure that, additionally to returning a numerical lower bound, provides a reason for it, i.e., a (presumably short) clause whose violation is a sufficient condition for the computed lower bound, see [MS00, MS04].

The second contribution of the paper is the connection between a $\texttt{DPLL}_{\texttt{BB}}$ execution and a proof of optimality. We show that each time that $\texttt{DPLL}_{\texttt{BB}}$ backjumps due to a soft conflict (i.e. the lower bound indicates that it is useless to extend the current assignment) we can infer a cost-based lemma, which is entailed from the problem. By recording these lemmas (among others), we can construct a very intuitive optimality proof.

This work could have been cast into the framework of SAT Modulo Theories (SMT) with a sequence of increasingly stronger theories [NO06]. However, the generation of proofs for SMT with theory strengthening has not been worked out (although the generation of unsatisfiable cores for normal SMT was analyzed in [CGS07]), and would in any case obfuscate the simple concept of proof we have here. Also, we believe that in its current form, the way we have integrated the concepts of lower bounding and cost-based propagation and learning is far more useful and accessible to a much wider audience.

This paper is structured as follows. In Section 2 we give some basic notions and preliminary definitions. In Section 3 the $\texttt{DPLL}_{\texttt{BB}}$ procedure is presented, whereas in Section 4 we develop the framework for the generation of proof certificates. Section 5 shows several important instances of problems that can be handled with $\texttt{DPLL}_{\texttt{BB}}$. Finally Section 6 gives conclusions of this work and points out directions for future research.

2 Preliminaries

We consider a fixed set of Boolean variables $\{x_1, \ldots, x_n\}$. *Literals*, denoted by the (subscripted, primed) letter l are elements of the set $\{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$. A *clause* (denoted by the letters $C, D, \ldots$) is a disjunction of literals $l_1 \vee \ldots \vee l_m$. The empty clause will be noted $\square$. A (partial truth) *assignment* I is a set of literals such that $\{x, \neg x\} \subseteq I$ for no x. A literal l is *true* in I if $l \in I$, *false* in I if $\neg l \in I$, and *undefined* in I otherwise. A clause C is true in I if at least one of its literals is true in I, false in I if all its literals are false in I, and undefined in I otherwise. Note that the empty clause is false in every assignment I. Sometimes we will write $\neg I$ to denote the clause that is the disjunction of the negations of the literals in I. A clause set S is true in I if all its clauses are true in I. Then I is called a *model* of S, and we write $I \models S$ (and similarly if a literal or clause is true in I). We sometimes write $I \models \neg C$ to indicate that all literals of a clause C are false in I.

We consider the following class of problems, which covers a broad spectrum of instances (see Section 5):

Definition 1. *A* Boolean optimization problem *is a pair* $(S, cost)$*, where* S *is a clause set, cost is a function* $cost\colon \mathbb{B}^n \to \mathbb{R}$*, and the goal is to find a model* A *of* S *such that* $cost(A)$ *is minimized.*

Definition 2. *A* cost clause *is an expression of the form* $C \vee c \geq k$ *where* C *is a clause and* $k \in \mathbb{R}$*.*

A cost clause $C \vee c \geq k$ may be better understood with its equivalent notation $\neg C \longrightarrow c \geq k$ which tells that if C is falsified, then the cost function must be greater than or equal to k.

Definition 3. *Let* $(S, cost)$ *be an optimization problem. A cost clause* $C \vee c \geq k$ *is* entailed by $(S, cost)$ *if* $cost(A) \geq k$ *for every model* A *of* S *such that* $A \models \neg C$*.*

Definition 4. *Given an optimization problem* $(S, cost)$*, a real number* k *is called a* lower bound *for an assignment* I *if* $cost(A) \geq k$ *for every model* A *of* S *such that* $I \subseteq A$*.*

A lower bounding procedure *lb is a procedure that, given an assignment* I*, returns a lower bound* k*, denoted* $lb(I)$*, and a cost clause of the form* $C \vee c \geq k$*, called the* lb-reason *of the lower bound, such that* $C \vee c \geq k$ *is entailed by* $(S, cost)$ *and* $I \models \neg C$*.*

Any procedure that can compute a lower bound k for a given I can be extended to a lower bounding procedure: it suffices to generate $\neg I \vee c \geq k$ as the lb-reason. But generating *short* lb-reasons is important for efficiency reasons, and in Section 5 we will see how this can be done for several classes of lower bounding methods.

3 Abstract Branch and Bound

3.1 $\texttt{DPLL}_{\texttt{BB}}$ Procedure

The $\texttt{DPLL}_{\texttt{BB}}$ procedure is modeled by a transition relation, defined by means of rules over states.

Definition 5. *A* $\texttt{DPLL}_{\texttt{BB}}$ state *is a 4-tuple* $I \parallel S \parallel k \parallel A$*, where:*
I *is a sequence of literals representing the current partial assignment,*
S *is a finite set of classical clauses (i.e. not cost clauses),*
k *is a real number representing the best-so-far cost,*
A *is the best-so-far model of* S *(i.e.* $cost(A) = k$*).*
Some literals l *in* I *are annotated as* decision literals *and written* l^{d}*.*

Note that the *cost* function and the variable set are not part of the states, since they do not change over time (they are fixed by the context).

Definition 6. *The* $\mathtt{DPLL}_{\mathtt{BB}}$ system *consists of the following rules:*

Decide :

$$I \parallel S \parallel k \parallel A \implies I\ l^{\mathsf{d}} \parallel S \parallel k \parallel A \quad \textbf{if} \quad \{\, l \text{ is undefined in } I$$

UnitPropagate :

$$I \parallel S \parallel k \parallel A \implies I\ l \parallel S \parallel k \parallel A \quad \textbf{if} \quad \begin{cases} C \vee l \in S,\ I \models \neg C \\ l \text{ is undefined in } I \end{cases}$$

Optimum :

$$I \parallel S \parallel k \parallel A \implies \mathit{OptimumFound} \quad \textbf{if} \quad \begin{cases} C \in S,\ I \models \neg C \\ \text{no decision literals in } I \end{cases}$$

Backjump :

$$I\ l^{\mathsf{d}}\ I' \parallel S \parallel k \parallel A \implies I\ l' \parallel S \parallel k \parallel A \quad \textbf{if} \quad \begin{cases} C \vee l' \in S,\ I \models \neg C \\ l' \text{ is undefined in } I \end{cases}$$

Learn :

$$I \parallel S \parallel k \parallel A \implies I \parallel S, C \parallel k \parallel A \quad \textbf{if} \quad \{\, (S, \mathit{cost}) \text{ entails } C \vee c \geq k$$

Forget :

$$I \parallel S, C \parallel k \parallel A \implies I \parallel S \parallel k \parallel A \quad \textbf{if} \quad \{\, (S, \mathit{cost}) \text{ entails } C \vee c \geq k$$

Restart :

$$I \parallel S \parallel k \parallel A \implies \emptyset \parallel S \parallel k \parallel A$$

Improve :

$$I \parallel S \parallel k \parallel A \implies I \parallel S \parallel k' \parallel I \quad \textbf{if} \quad \{\, I \models S \text{ and } \mathit{cost}(I) = k' < k$$

As we will see, one can use these rules for finding an optimal solution to a problem (S, cost) by generating an arbitrary derivation $\emptyset \parallel S \parallel \infty \parallel \emptyset \implies \ldots$. It will always terminate with $\ldots \implies I \parallel S' \parallel k \parallel A \implies \mathit{OptimumFound}$. Then A is a minimum-cost model for S with $\mathit{cost}(A) = k$. If S has no models at all then A will be $\emptyset$ and $k = \infty$.

All the rules except Improve are natural extensions of the Abstract DPLL approach of [NOT06]. In the following we briefly explain them.

- The Decide rule represents a case split: an undefined literal l is chosen and added to I, annotated as a decision literal.
- UnitPropagate forces a literal l to be true if there is a clause $C \vee l$ in S whose part C is false in I.
- The Optimum rule expresses that if in a state $I \parallel S \parallel k \parallel A$ in S there is a so-called *conflicting clause* C (i.e., such that $I \models \neg C$), and there is no decision literal in I, then the optimization procedure has terminated, which shows that the best-so-far cost is optimal.
- On the other hand, if there is some decision literal in I and an entailed conflicting clause, then one can always find (and Learn) a *backjump clause*, an entailed cost clause of the form $C \vee l' \vee c \geq k$, such that Backjump using $C \vee l'$ applies (see Lemma 1 below). Good backjump clauses can be

found by *conflict analysis* of the conflicting clause [MSS99, ZMMM01], see Example 3.2 below.
- By Learn one can add any entailed cost clause to S. Learned clauses prevent repeated work in *similar* conflicts, which frequently occur in industrial problems having some regular structure. Notice that when such a clause is learned the $c \geq k$ literal is dropped (it is only kept at a metalevel for the generation of optimality certificates, see Section 4).
- Since a lemma is aimed at preventing future similar conflicts, it can be removed using Forget, when such conflicts are not very likely to be found again. In practice this is done if its *activity*, that is, how many times it has participated in *recent* conflicts, has become low.
- Restart is used to escape from bad search behaviors. The newly learned clauses will lead the heuristics for Decide to behave differently, and hopefully make $\mathtt{DPLL}_{\mathtt{BB}}$ explore the search space in a more compact way.
- Improve allows one to model non-trivial optimization concepts, namely *cost-based backjumping* and *and cost-based propagation*. If $lb(I) \geq k$, the lower bounding procedure can provide an *lb*-reason $C \vee c \geq k$. As explained above, given this conflicting clause, Backjump applies if there is some decision literal in I, and otherwise Optimum is applicable. A *cost-based propagation* of a literal l that is undefined in I can be made if $lb(I\,l) \geq k$ ([XZ05]; for linear cost functions, cf. the "limit lower bound theorem" of [CM95]). Then again the corresponding *lb*-reason is conflicting and either Backjump or Optimum applies.

Lemma 1. *(See [NOT06] for proofs of this and other related properties.) Let* $(S, cost)$ *be an optimization problem, and assume*

$$\emptyset \parallel S \parallel \infty \parallel \emptyset \implies \ldots \implies I \parallel S' \parallel k \parallel A$$

If there is some decision literal in I *and* C *is entailed by* $(S', cost)$ *and conflicting in* I*, then* I *is of the form* $I'\, l^{\mathsf{d}}\, I''$ *and there exists a backjump clause, i.e., a cost clause of the form* $C \vee l' \vee c \geq k$ *that is entailed by* $(S', cost)$ *and such that* $I' \models \neg C$ *and* l' *is undefined in* I'*.*

The potential of the previous rules will be illustrated in Section 3.2. The correctness of $\mathtt{DPLL}_{\mathtt{BB}}$ is summarized in Theorem 1:

Definition 7. *A derivation* $\emptyset \parallel S \parallel \infty \parallel \emptyset \implies \ldots$ *is* progressive *if it contains only finitely many consecutive* Learn *or* Forget *steps and* Restart *is applied with increasing periodicity.*

Theorem 1. *Let* $(S, cost)$ *be an optimization problem, and consider a progressive derivation with initial state* $\emptyset \parallel S \parallel \infty \parallel \emptyset$*. Then this derivation is finite. Moreover, if a* final state *is reached, i.e., a state to which no rule can be applied, then the derivation is of the form*

$$\emptyset \parallel S \parallel \infty \parallel \emptyset \implies \ldots \implies I \parallel S' \parallel k \parallel A \implies \mathit{OptimumFound}$$

and then A *is a minimum-cost model for* S*, where* $cost(A) = k$*. In particular,* S *has no models if, and only if,* $k = \infty$ *and* $A = \emptyset$*.*

Of course the previous formal result provides more freedom in the strategy for applying the rules than needed. Practical implementations will only generate progressive derivations. Typically at each conflict the backjump clause is learned, and from time to time a certain portion of the learned clauses is forgotten (e.g., the 50% of less active ones). Restarts are applied with increasing periodicity by, e.g., restarting after a certain number N of conflicts and then increasing N.

3.2 $\mathtt{DPLL_{BB}}$ Example

Consider the clause set S defined over $x_1, \ldots x_6$ (denoting $\neg x_i$ by $\bar{x}_i$):

1.	$x_2 \vee x_4$	5.	$x_1 \vee \bar{x}_3 \vee \bar{x}_6$
2.	$x_2 \vee \bar{x}_5$	6.	$\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_6$
3.	$x_4 \vee \bar{x}_5$	7.	$x_2 \vee x_3 \vee x_5 \vee \bar{x}_6$
4.	$x_5 \vee x_6$	8.	$x_2 \vee \bar{x}_3 \vee x_5 \vee \bar{x}_6$

where $cost(x_1, \ldots x_6) = 1x_1 + 2x_2 + \ldots + 6x_6$, i.e., subindices are cost coefficients. We start an $\mathtt{DPLL_{BB}}$ derivation, first deciding x_6 to be false (setting high-cost variables to false can be a good heuristic):

$$
\begin{array}{lll}
 & \emptyset & \| S \| \infty \| \emptyset \\
\Longrightarrow_{\mathsf{Decide}} & \bar{x}_6^{\mathsf{d}} & \| S \| \infty \| \emptyset \\
\Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_6^{\mathsf{d}} x_5 & \| S \| \infty \| \emptyset \\
\Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_6^{\mathsf{d}} x_5 x_2 & \| S \| \infty \| \emptyset \\
\Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_6^{\mathsf{d}} x_5 x_2 x_4 & \| S \| \infty \| \emptyset \\
\Longrightarrow_{\mathsf{Decide}} & \bar{x}_6^{\mathsf{d}} x_5 x_2 x_4 \bar{x}_3^{\mathsf{d}} & \| S \| \infty \| \emptyset \\
\Longrightarrow_{\mathsf{Decide}} & \bar{x}_6^{\mathsf{d}} x_5 x_2 x_4 \bar{x}_3^{\mathsf{d}} \bar{x}_1^{\mathsf{d}} & \| S \| \infty \| \emptyset
\end{array}
$$

Now, since $\bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1$ is a model of S of cost $11 < \infty$, we can apply $\mathsf{Improve}$ and the corresponding *lb*-reason, e.g., $\bar{x}_2 \vee \bar{x}_4 \vee \bar{x}_5 \vee c \geq 11$, then becomes a conflicting clause. Intuitively, it expresses that any assignment where x_2, x_4 and x_5 are set to true must have cost at least 11. Now, a *conflict analysis* procedure starting form this conflicting clause can be used to compute a backjump clause. This is done by successive resolution steps on the conflicting clause, resolving away the literals $\bar{x}_4$ and $\bar{x}_2$ in the reverse order their negations were propagated, with the respective clauses that caused the propagations:

$$
\dfrac{\dfrac{\bar{x}_2 \vee \bar{\mathbf{x}}_{\mathbf{4}} \vee \bar{x}_5 \vee c \geq 11 \quad \mathbf{x_4} \vee \bar{x}_5}{\bar{\mathbf{x}}_{\mathbf{2}} \vee \bar{x}_5 \vee c \geq 11} \qquad \mathbf{x_2} \vee \bar{x}_5}{\bar{x}_5 \vee c \geq 11}
$$

until a single literal of the current decision level (called the *1UIP*) is left, yielding $\bar{x}_5 \vee c \geq 11$. Learning the clause $C = \bar{x}_5$ allows one to jump from decision level 3 back to decision level 0 and assert x_5. All this can be modeled as follows:

$$
\begin{array}{lll}
\ldots \Longrightarrow_{\mathsf{Improve}} & \bar{x}_6^{\mathsf{d}} x_5 x_2 x_4 \bar{x}_3^{\mathsf{d}} \bar{x}_1^{\mathsf{d}} & \| S \| 11 \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\
\Longrightarrow_{\mathsf{Learn}} & \bar{x}_6^{\mathsf{d}} x_5 x_2 x_4 \bar{x}_3^{\mathsf{d}} \bar{x}_1^{\mathsf{d}} & \| S, C \| 11 \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\
\Longrightarrow_{\mathsf{Backjump}} & \bar{x}_5 & \| S, C \| 11 \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1
\end{array}
$$

Now the derivation could continue, e.g., as follows:

$$\begin{array}{lllll} \ldots & \Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_5 x_6 & \| S, C \ \| 11 & \| \ \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\ & \Longrightarrow_{\mathsf{Decide}} & \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} & \| S, C \ \| 11 & \| \ \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\ & \Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} x_2 & \| S, C \ \| 11 & \| \ \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \end{array}$$

Now notice that x_3 is not assigned, and that since x_2 and x_6 are true in the current partial assignment any assignment strictly improving the best-so-far cost 11 must assign x_3 to false. As explained above, this cost-based propagation can be modeled as follows. The lower bounding procedure expresses the fact that any solution setting x_2, x_3 and x_6 to true has cost no better than 11 by means of the *lb*-reason $\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_6 \vee c \geq 11$. This is an entailed cost clause that is learned as $C' = \bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_6$. Then literal $\bar{x}_3$ is propagated.

$$\begin{array}{lllll} \ldots & \Longrightarrow_{\mathsf{Learn}} & \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} x_2 & \| S, C, C' \ \| 11 & \| \ \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\ & \Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} x_2 \bar{x}_3 & \| S, C, C' \ \| 11 & \| \ \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \end{array}$$

If we now UnitPropagate x_1 with clause 5, clause 6 becomes conflicting. As usual, a backjump clause is computed by doing conflict analysis from the falsified clause, using among others the clause C' that was learned to propagate $\bar{x}_3$:

$$\dfrac{\dfrac{\mathbf{x_1} \vee x_3 \vee \bar{x}_6 \quad \bar{\mathbf{x}}_\mathbf{1} \vee x_3 \vee \bar{x}_6}{\mathbf{x_3} \vee \bar{x}_6} \qquad \bar{x}_2 \vee \bar{\mathbf{x}}_\mathbf{3} \vee \bar{x}_6 \vee c \geq 11}{\bar{x}_2 \vee \bar{x}_6 \vee c \geq 11}$$

Learning $C'' = \bar{x}_2 \vee \bar{x}_6$ allows one to jump back to decision level 0 asserting $\bar{x}_2$.

$$\begin{array}{lllll} \ldots & \Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} x_2 \bar{x}_3 x_1 & \| S, C, C' & \| 11 \ \| \ \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\ & \Longrightarrow_{\mathsf{Learn}} & \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} x_2 \bar{x}_3 x_1 & \| S, C, C', C'' & \| 11 \ \| \ \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\ & \Longrightarrow_{\mathsf{Backjump}} & \bar{x}_5 x_6 \bar{x}_2 & \| S, C, C', C'' & \| 11 \ \| \ \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \end{array}$$

Finally after unit propagating with clause 7 one gets a conflict with clause 8, and as no decision literals are left, the optimization procedure terminates:

$$\begin{array}{lllll} \ldots & \Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_5 x_6 \bar{x}_2 x_3 & \| S, C, C', C'' & \| 11 \ \| \ \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\ & \Longrightarrow_{\mathsf{Optimum}} & \mathit{OptimumFound} & & \end{array}$$

□

4 Certificates of Optimality

In the following, we show how from a certain trace of an $\mathtt{DPLL}_{\mathtt{BB}}$ execution one can extract a formal proof of optimality in a proof system asserting "A is an optimal model of S with respect to *cost*". Our proof system relies on the following type of resolution over cost clauses,

Definition 8. *The* Cost Resolution *rule is the following inference rule with two cost clauses as premises and another cost clause as conclusion:*

$$\dfrac{x \vee C \vee c \geq k \qquad \neg x \vee D \vee c \geq k'}{C \vee D \vee c \geq min(k, k')} \qquad \text{Cost Resolution}$$

Cost Resolution behaves like classical resolution, except in that it further exploits the fact that $c \geq k \vee c \geq k'$ is equivalent to $c \geq min(k, k')$. In what follows, when needed a clause C from S will be seen as the trivially entailed cost clause $C \vee c \geq \infty$.

Theorem 2. *Cost Resolution is* correct, *that is, if* $x \vee C \vee c \geq k$ *and* $\neg x \vee D \vee c \geq k'$ *are cost clauses entailed by an optimization problem* $(S, cost)$*, then* $C \vee D \vee c \geq min(k, k')$ *is also entailed by* $(S, cost)$*.*

Definition 9. *Let* S *be a set of cost clauses and let* C *be a cost clause. A* Cost Resolution proof *of* C *from* S *is a binary tree where:*

- *each node is (labeled by) a cost clause*
- *the root is* C
- *the leaves are clauses from* S
- *every non-leaf node has two parents from which it can be obtained in one Cost Resolution step.*

Together with a model A such that $cost(A) = k$, a *k-lower-bound certificate* as we define now gives a precise k-optimality certificate for $(S, cost)$:

Definition 10. *A* k-lower-bound certificate *for an optimization problem* $(S, cost)$ *consists of the following three components:*

1. *a set of cost clauses* S'
2. *a Cost-Resolution Proof of the clause* $c \geq k$ *from* $S \cup S'$
3. *for each cost clause in* S'*, a proof of entailment of it from* $(S, cost)$

As we will see, the set of cost clauses S' of component 1. of this definition corresponds to the different *lb*-reasons generated by the lower bounding procedure that may have been used along the $\texttt{DPLL}_{\texttt{BB}}$ derivation. A very simple independent k-lower-bound certificate checker can just check the cost resolution proof, *if the lower bounding procedure is trusted in that indeed all cost clauses of* S' *are entailed*. Then, since by correctness of Cost Resolution the root $c \geq k$ of a Cost Resolution proof is entailed if the leaves are entailed, a k-lower-bound certificate guarantees that $c \geq k$ is indeed entailed by $(S \cup S', cost)$, and the entailment of $c \geq k$ by definition means that "$cost(A) \geq k$ for every model A of S".

If one cannot trust the lower bounding procedure, then also component 3. is needed. The notion of a "proof of entailment" from $(S, cost)$ for each cost clause in S' of course necessarily depends on the particular lower bounding procedure used, and an independent optimality proof checker should hence have some knowledge of the deductive process used by the lower bounding procedure. This aspect is addressed in detail in the next section.

4.1 Generation of *k*-Lower-Bound Certificates

Each time an *lb*-reason is generated and used in an $\texttt{DPLL}_{\texttt{BB}}$ execution, it is written to a file which we will call S' since it corresponds to component 1. of the k-lower-bound certificate. Now observe that any execution of $\texttt{DPLL}_{\texttt{BB}}$ terminates with a

step of Optimum, i.e., with a conflict at decision level 0. From a standard SAT solver point of view, this means that $S \cup S'$ forms an unsatisfiable SAT instance and a refutation proof for this contradiction can be reconstructed as follows (cf. [ZM03] for details). All clauses in S and in S' get a unique identifier (ID). Each time a backjump step takes place, the backjump clause also gets a (unique) ID and a line `ID ID1 ... IDm` is written to a trace file, where `ID1 ... IDm` are the IDs of all parent clauses in the conflict analysis process generating this backjump clause. A last line is written when the conflict at decision level 0 is detected for the parents of this last conflict analysis which produces the empty clause. By processing backwards this trace file, composing all the component resolution proofs from each conflict analysis, a resolution proof from $S \cup S'$ of the last derived clause, i.e., the empty clause, can be constructed.

If we recover the cost literals of cost clauses (recall that the Learn rule of $\texttt{DPLL}_{\texttt{BB}}$ drops the cost literal) in the refutation proof, it turns out that it becomes a k-lower-bound certificate where k is the optimum of the problem. The reason is that in a Cost Resolution proof the cost literal of the root clause is the minimum among the cost literals of the leaf clauses. The following example illustrates the whole process.

Example 1. For the $\texttt{DPLL}_{\texttt{BB}}$ derivation of Section 3.2, the initial clauses have ID's `1-8`, the set the S' will contain the the *lb*-reasons $\bar{x}_2 \vee \bar{x}_4 \vee \bar{x}_5 \vee c \geq 11$ and $\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_6 \vee c \geq 11$ with ID's `9` and `10` respectively. The two backjump clauses $\bar{x}_5 \vee c \geq 11$ and $\bar{x}_2 \vee \bar{x}_6 \vee c \geq 11$ and the final "empty" clause $c \geq 11$ get ID's `11,12,13` respectively, and the trace file will be:

$$\begin{aligned} &\texttt{11} \leftarrow \texttt{2,3,9} \\ &\texttt{12} \leftarrow \texttt{5,6,10} \\ &\texttt{13} \leftarrow \texttt{4,7,8,11,12} \end{aligned}$$

By processing this file backwards it is straightforward to produce a Cost Resolution proof of $c \geq 11$. This is done below, where for lack of space the proof has been split in two at the clause marked with $(*)$. This proof, together with each *lb*-reason and its entailment certificate, will constitute an 11-lower-bound certificate. The optimality certificate is finally obtained with the addition of the 11-upper-bound certificate $\bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1$.

$$\dfrac{\dfrac{x_2 \vee x_3 \vee x_5 \vee \bar{x}_6 \quad x_2 \vee \bar{x}_3 \vee x_5 \vee \bar{x}_6}{x_2 \vee x_5 \vee \bar{x}_6} \qquad \dfrac{\dfrac{x_1 \vee x_3 \vee \bar{x}_6 \quad \bar{x}_1 \vee x_3 \vee \bar{x}_6}{x_3 \vee \bar{x}_6} \qquad \bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_6 \vee c \geq 11}{\bar{x}_2 \vee \bar{x}_6 \vee c \geq 11}}{x_5 \vee \bar{x}_6 \vee c \geq 11 \ (*)}$$

$$\dfrac{\dfrac{x_5 \vee x_6 \quad x_5 \vee \bar{x}_6 \vee c \geq 11 \ (*)}{x_5 \vee c \geq 11} \qquad \dfrac{\dfrac{\bar{x}_2 \vee \bar{x}_4 \vee \bar{x}_5 \vee c \geq 11 \quad x_4 \vee \bar{x}_5}{\bar{x}_2 \vee \bar{x}_5 \vee c \geq 11} \qquad x_2 \vee \bar{x}_5}{\bar{x}_5 \vee c \geq 11}}{c \geq 11}$$

5 Instances of DPLL_{BB} and Lower Bounding Procedures

In order to complete the method for generating optimality certificates, in this section we show, for different classes of cost functions, several lower bounding procedures together with ways for proving the entailment from $(S, cost)$ for any lb-reason $C \vee c \geq k$ they generate.

Of course a general approach for this is to provide a list of all the models $A_1, \ldots, A_m$ of $S \wedge \neg C$, checking that each one of them has cost at least k, together with a resolution refutation of $S \wedge \neg C \wedge \neg A_1 \wedge \cdots \neg A_m$, which shows that these $A_1, \ldots, A_m$ are indeed all the models of $S \wedge \neg C$.

But this will usually not be feasible in practice. Therefore, we now describe some lower bounding procedures producing simple and compact certificates that can be understood by *ad-hoc* proof checkers.

5.1 Linear Cost Functions

A very important class of optimization problems is that with *linear* cost functions, i.e., of the form $cost(x_1, \ldots, x_n) = \sum_{i=1}^{n} c_i x_i$ for certain $c_i \geq 0$. In this context c_i is called the *cost* of variable x_i. Note that this also covers the cases of negative costs or costs associated to negated variables, which are harmlessly reduced to this one.

Linear Boolean optimization has many applications, amongst others Automatic Test Pattern Generation [FNMS01], FPGA Routing, Electronic Design Automation, Graph Coloring, Artificial Intelligence Planning [HS00] and Electronic Commerce [San99]. In particular the case where $c_i = 1$ for all $1 \leq i \leq n$, called the *Min-Ones* problem, appears naturally in the optimization versions of important well-known NP-complete problems such as the maximum clique or the minimum hitting set problems.

The problem of computing lower bounds for linear optimization problems in a branch-and-bound setting has been widely studied in the literature. Here we consider the two main techniques for that purpose: *independent sets* and *linear programming*.

Independent Sets. Given a partial assignment I and a clause C, let $undef_I(C)$ denote the set of literals in C which are undefined in I, i.e., $undef_I(C) = \{l \in C \mid l \notin I \text{ and } \neg l \notin I\}$. A set of clauses M is an *independent set for* I if:

- for all $C \in M$, neither $I \models C$ nor $I \models \neg C$;
- for all $C \in M$, $undef_I(C)$ is non-empty and only contains positive literals;
- for all $C, C' \in M$ such that $C \neq C'$, $undef_I(C) \cap undef_I(C') = \emptyset$.

If M is an independent set for I, any total assignment extending I and satisfying M has cost at least

$$K = \sum_{x_i \in I} c_i + \sum_{C \in M} \min\{c_j \mid x_j \in C \text{ and } \neg x_j \notin I\}$$

since satisfying each clause C of M will require to add the minimum cost of the positive non-false (in I) literals in C. Independent sets have been used in

e.g., [Cou96, MS02]. In [FM06] they are precomputed in order to speed up the actual branch-and-bound procedure.

In this case the lower bounding procedure generates the *lb*-reason $\neg I' \vee c \geq K$, where $I' \subseteq I$ contains:

- the positive literals in I with non-null cost;
- the positive literals whose negations appear in M (which belong to I); and
- the negative literals $\neg x_i \in I$ such that $x_i \in C$ for some $C \in M$ and $c_i < \min\{c_j \mid x_j \in C \text{ and } \neg x_j \notin I\}$.

For this lower bounding procedure a proof of entailment of the *lb*-reason must of course contain the independent set M itself. Then the proof checker can check that $M \subseteq S$, that M is indeed independent for I and that $K \geq k$.

Example 2. Consider the clause set $S = \{\, x_1 \vee x_3 \vee x_5,\ x_2 \vee x_4 \vee x_5 \vee \neg x_6,\ \neg x_1 \vee \neg x_2 \,\}$, and the cost function $cost(x_1, x_2, x_3, x_4) = \sum_{i=1}^{6} i \cdot x_i$. It is easy to see that $M = \{x_1 \vee x_3 \vee x_5,\ x_2 \vee x_4 \vee x_5 \vee \neg x_6\}$ is an independent set for the partial assignment $I = \{\neg x_5, x_6\}$. The lower bound is $6 + \min(1, 3) + \min(2, 4) = 9$, and the *lb*-reason $x_5 \vee \neg x_6 \vee c \geq 9$ is produced. □

Linear Programming [LD97, Li04]**.** This approach for computing lower bounds relies on the fact that linear Boolean optimization is a particular case of 0-1 Integer Linear Programming. Indeed, such a Boolean optimization problem can be transformed into an integer program by imposing for each variable x that $0 \leq x \leq 1$ and $x \in \mathbb{Z}$, and transforming each clause $x_1 \vee \cdots \vee x_n \vee \neg y_1 \vee \cdots \vee \neg y_m$ into the linear constraint $\sum_{i=1}^{n} x_i + \sum_{j=1}^{m} (1 - y_j) \geq 1$. The current partial assignment I is encoded by imposing additional constraints $x = 1$ if $x \in I$, $x = 0$ if $\neg x \in I$. Then a lower bound can be computed by dropping the integrality condition and solving the resulting relaxation in the rationals with an LP solver.

If K is the lower bound obtained after solving the relaxation, an *lb*-reason of the form $\neg I' \vee c \geq K$ where $I' \subseteq I$ can be computed using an exact dual solution of multipliers [Sch86] (which may be computed by an exact LP solver [Mak08]). Moreover, a proof of entailment of this *lb*-reason consists in the dual solution itself, which proves the optimality of K.

Example 3. Consider again the clause set, the cost function and the partial assignment as in Example 2. In this case the linear program is

$\min\{x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6 \mid x_1 + x_3 + x_5 \geq 1,\ x_2 + x_4 + x_5 - x_6 \geq 0,$
$-x_1 - x_2 \geq -1,\ x_5 = 0,\ x_6 = 1,\ 0 \leq x_1, x_2, x_3, x_4 \leq 1\}$, whose optimum is 11. A proof of optimality (in fact, of the lower bound) is:

$$
\begin{array}{rcccccccccccc}
 & & x_1 & +2\,x_2 & +3\,x_3 & +4\,x_4 & +5\,x_5 & +6\,x_6 & -11 & = \\
+3 & (& x_1 & & + & x_3 & & + & x_5 & & -1 &) \\
+4 & (& & x_2 & & & + & x_4 + & x_5 - & x_6 & &) \\
+2 & (& -x_1 - & x_2 & & & & & & +1 & &) \\
-2 & & & & & & x_5 & & & & & \\
+10 & (& & & & & & & & x_6 & -1 &)
\end{array}
$$

which witnesses that $x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6 \geq 11$ for all x_1, x_2, x_3, x_4, x_5, x_6 such that $x_1 + x_3 + x_5 \geq 1$, $x_2 + x_4 + x_5 - x_6 \geq 0$, $-x_1 - x_2 \geq -1$,

$x_5 \leq 0$ and $x_6 \geq 1$. This can be used as a proof of entailment of the *lb*-reason $x_5 \vee \neg x_6 \vee c \geq 11$ (notice that none of the literals of the assignment is dropped in the *lb*-reason since both $x_5 \leq 0$ and $x_6 \geq 1$ are used). □

5.2 Max-SAT

In a (partial weighted) Max-SAT problem $(S, cost)$, the cost function is defined by a set of so-called *soft* clauses S' with a *weight* function $\omega : S' \to \mathbb{R}$. Then the cost of a total assignment A is the sum of the weights of the clauses in S' that are false in A. Note that S' is disjoint from the (possibly empty) set of clauses S, which are called *hard* clauses in this context. Max-SAT has many applications, among others Probabilistic Reasoning [Par02], Frequency Assignment [RLFAP99], Computer Vision, Machine Learning and Pattern Recognition (see the introduction of [Wer05]).

Max-SAT as a non-linear polynomial cost function. Given a clause $C = y_1 \vee \ldots \vee y_p \vee \neg z_1 \vee \ldots \vee \neg z_q$ over a set of variables $\{x_1 \ldots x_n\}$, the polynomial $p_C(x_1, \ldots, x_n) = \prod_{i=1}^{p}(1 - y_i) \cdot \prod_{j=1}^{m}(z_j)$ fulfills for any total assignment A that $p_C(A) = 1$ if $A \models \neg C$, and $p_C(A) = 0$ otherwise. Therefore we have that $cost(A) = \sum_{C \in S'} p_C(A) \cdot \omega(C)$.

Linear Boolean optimization vs Max-SAT. Linear Boolean optimization can be cast as an instance of Max-SAT by having one soft unit positive clause for each variable with non-null cost, with this cost as weight. Reciprocally, Max-SAT can be expressed as a linear optimization problem by adding slack variables to soft clauses. However, this translation is normally unpractical, making the SAT solver extremely slow, since, e.g., it hinders the application of unit propagation [ANORC08].

Branch and bound for Max-SAT. But most of the research in recent years in the Max-SAT community has been devoted to the computation of good quality lower bounds to be used within a branch-and-bound setting. As shown in [LHdG08], most of these lower bounding procedures can be seen as limited forms of *Max-resolution* (see below). Since Max-resolution is sound, theoretically one can in fact use it to certify optimality in any Max-SAT problem. But the growth in the number of clauses makes this unpractical except for small problems. However, one can use it for the proof of entailment for individual *lb*-reasons.

For simplicity, we show here Max-resolution for soft clauses of the form $(l_1 \vee l_2, w)$, where w denotes the weight:

$$\frac{(x \vee a, u) \quad (\neg x \vee b, v)}{(a \vee b, m)(x \vee a, u - m)(\neg x \vee b, v - m)(x \vee a \vee \neg b, m)(\neg x \vee b \vee \neg a, m)}$$

where $m = \min(u, v)$ and the conclusions replace the premises instead of being added to the clause set.

Example 4. Consider a Max-SAT problem without hard clauses and where soft clauses are $S' = \{ (x_1 \vee x_2 \vee x_3, 1), (x_1 \vee \neg x_2 \vee x_3, 2), (\neg x_1 \vee x_2 \vee x_3, 3), (\neg x_1 \vee$

$\neg x_2 \vee x_3, 4)$ }. Given the partial assignment $I = \{\neg x_3, x_4\}$, by means of the following steps of Max-resolution

$$\dfrac{\dfrac{(x_1 \vee \mathbf{x_2} \vee x_3, 1) \quad (x_1 \vee \neg\mathbf{x_2} \vee x_3, 2)}{\vdots\ (\mathbf{x_1} \vee x_3, 1)} \quad \dfrac{(\neg x_1 \vee \mathbf{x_2} \vee x_3, 3) \quad (\neg x_1 \vee \neg\mathbf{x_2} \vee x_3, 4)}{\vdots\ (\neg\mathbf{x_1} \vee x_3, 3)}}{(x_3, 1)}$$

one gets clause x_3 with weight 1. Taking into account the partial assignment $I = \{\neg x_3, x_4\}$, this clause implies that 1 is a lower bound and a *lb*-reason is $\neg x_3 \vee c \geq 1$. Moreover, the proof of Max-resolution above proves the entailment of the *lb*-reason. □

6 Conclusions

Our abstract DPLL-based branch-and-bound algorithm, although being very similar to abstract DPLL, can model optimization concepts such as cost-based propagation and cost-based learning. Thus, $\mathtt{DPLL}_{\mathtt{BB}}$ is natural to SAT practitioners, but still faithful to most state-of-the-art branch-and-bound solvers. Interestingly, several branch-and-bound solvers, even state-of-the-art ones, still do not use cost-based backjumping and propagation, which appear naturally in $\mathtt{DPLL}_{\mathtt{BB}}$. Our formal definition of optimality certificates and the description of how a $\mathtt{DPLL}_{\mathtt{BB}}$ trace can be used to generate them turns out to be elegant and analagous to the generation of refutation proofs by resolution in SAT.

We think that $\mathtt{DPLL}_{\mathtt{BB}}$ will help understanding and reasoning about new branch-and-bound implementations and extensions. For example, it is not difficult to use it for computing the m *best* (i.e., lowest-cost) models for some m, or for computing all models with cost lower than a certain threshold, and also the certificates for these can be derived without much effort.

References

[ANORC08] Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Efficient Generation of Unsatisfiability Proofs and Cores in SAT. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS, vol. 5330, pp. 16–30. Springer, Heidelberg (2008)

[RLFAP99] Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.P.: Radio Link Frequency Assignment. Constraints 4, 79–89 (1999)

[CGS07] Cimatti, A., Griggio, A., Sebastiani, R.: A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 334–339. Springer, Heidelberg (2007)

[CM95] Coudert, O., Madre, J.C.: New Ideas for Solving Covering Problems. In: Proc. of DAC 1995, pp. 641–646. ACM, New York (1995)

[Cou96] Coudert, O.: On Solving Binate Covering Problems. In: Proc. of DAC 1996, pp. 197–202. ACM, New York (1996)

[FM06] Fu, Z., Malik, S.: Solving the Minimum Cost Satisability Problem Using SAT Based Branch-and-Bound Search. In: Proc. of ICCAD 1996, pp. 852–859 (2006)

[FNMS01] Flores, P.F., Neto, H.C., Marques-Silva, J.P.: An Exact Solution to the Minimum Size Test Pattern Problem. ACM Trans. Des. Autom. Electron. Syst. 6(4), 629–644 (2001)

[HS00] Hoos, H.H., Stützle, T.: SATLIB: An Online Resource for Research on SAT. In: Proc. of SAT 2000, pp. 283–292. IOS Press, Amsterdam (2000), www.satlib.org

[LD97] Liao, S., Devadas, S.: Solving Covering Problems Using LPR-Based Lower Bounds. In: Proc. of DAC 1997, pp. 117–120. ACM, New York (1997)

[LHdG08] Larrosa, J., Heras, F., de Givry, S.: A Logical Approach to Efficient Max-SAT Solving. Artif. Intell. 172(2-3), 204–233 (2008)

[Li04] Li, X.Y.: Optimization Algorithms for the Minimum-Cost Satisfiability Problem. PhD thesis, Dept. Comp. Sc., N. Carolina State Univ. (2004)

[Mak08] Makhorin, A.: GNU Linear Programming Kit (2008), http://www.gnu.org/software/glpk/glpk.html

[MS00] Manquinho, V.M., Marques Silva, J.P.: Search Pruning Conditions for Boolean Optimization. In: Proc. of ECAI 2000, pp. 103–107. IOS Press, Amsterdam (2000)

[MS02] Manquinho, V.M., Marques Silva, J.P.: Search Pruning Techniques in SAT-Based Branch-and-bound Algorithms for the Binate Covering Problem. IEEE Trans. on CAD of Integ. Circ. and Syst. 21(5), 505–516 (2002)

[MS04] Manquinho, V.M., Marques Silva, J.P.: Satisfiability-Based Algorithms for Boolean Optimization. Ann. Math. Artif. Intell. 40(3-4), 353–372 (2004)

[MSS99] Marques-Silva, J., Sakallah, K.A.: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Trans. Comput. 48(5), 506–521 (1999)

[NO06] Nieuwenhuis, R., Oliveras, A.: On SAT Modulo Theories and Optimization Problems. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 156–169. Springer, Heidelberg (2006)

[NOT06] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)

[Par02] Park, J.D.: Using Weighted Max-SAT Engines to Solve MPE. In: Proc. of AAAI 2002, Edmonton, Alberta, Canada, pp. 682–687 (2002)

[San99] Sandholm, T.: An Algorithm for Optimal Winner Determination in Combinatorial Auctions. In: IJCAI 1999, pp. 542–547 (1999)

[Sch86] Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons, Chichester (1986)

[Wer05] Werner, T.: A Linear Programming Approach to Max-Sum Problem: A review. Technical Report CTU-CMP-2005-25, Center for Machine Perception, Czech Technical University (2005)

[XZ05] Xing, Z., Zhang, W.: Maxsolver: An Efficient Exact Algorithm for (Weighted) Maximum Satisfiability. Artif. Intell. 164(1-2), 47–80 (2005)

[ZM03] Zhang, L., Malik, S.: Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In: Proc. of DATE 2003, pp. 10880–10885. IEEE Computer Society, Los Alamitos (2003)

[ZMMM01] Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In: Proc. of ICCAD 2001, pp. 279–285 (2001)

Exploiting Cycle Structures in Max-SAT

Chu Min Li[1], Felip Manyà[2], Nouredine Mohamedou[1], and Jordi Planes[3,⋆]

[1] MIS, Université de Picardie Jules Verne, 5 Rue du Moulin Neuf 80000 Amiens, France
[2] IIIA-CSIC, Campus UAB, 08193 Bellaterra Spain
[3] Computer Science Department, Universitat de Lleida, Jaume II, 69, 25001 Lleida, Spain

Abstract. We investigate the role of cycles structures (i.e., subsets of clauses of the form $\bar{l}_1 \vee l_2, \bar{l}_1 \vee l_3, \bar{l}_2 \vee \bar{l}_3$) in the quality of the lower bound (LB) of modern MaxSAT solvers. Given a cycle structure, we have two options: (i) use the cycle structure just to detect inconsistent subformulas in the underestimation component, and (ii) replace the cycle structure with $\bar{l}_1, l_1 \vee \bar{l}_2 \vee \bar{l}_3, \bar{l}_1 \vee l_2 \vee l_3$ by applying MaxSAT resolution and, at the same time, change the behaviour of the underestimation component. We first show that it is better to apply MaxSAT resolution to cycle structures occurring in inconsistent subformulas detected using unit propagation or failed literal detection. We then propose a heuristic that guides the application of MaxSAT resolution to cycle structures during failed literal detection, and evaluate this heuristic by implementing it in MaxSatz, obtaining a new solver called MaxSatz$_c$. Our experiments on weighted MaxSAT and Partial MaxSAT instances indicate that MaxSatz$_c$ substantially improves MaxSatz on many hard random, crafted and industrial instances.

1 Introduction

The lower bound (LB) computation method implemented in branch and bound MaxSAT solvers (e.g. [4,6,10,12,13]) is decisive for obtaining a competitive solver. The LB of MaxSatz [10] and MiniMaxSat [4] —two of the best performing solvers in the 2008 MaxSAT Evaluation— has two components: (i) the *underestimation component*, which detects disjoint inconsistent subformulas and takes the number of detected subformulas as an underestimation of the LB, and (ii) the *inference component*, which applies inference rules and, in the best case, makes explicit a contradiction by deriving an empty clause which allows to increment the LB. Both components are applied at each node of the search space, and cooperate rather than work independently.

A MaxSAT instance may contain different structures that influence the behavior of the two components of the LB. In this paper we investigate the role of the so-called cycles structures (i.e., subsets of clauses of the form $\bar{l}_1 \vee l_2, \bar{l}_1 \vee l_3, \bar{l}_2 \vee \bar{l}_3$)[1] in the quality of the LB. Given a cycle structure, we have two options: (i) use the cycle structure just to detect inconsistent subformulas in the underestimation component

⋆ Research partially supported by the Generalitat de Catalunya under grant 2005-SGR-00093, and the *Ministerio de Ciencia e Innovación* research projects CONSOLIDER CSD2007-0022, INGENIO 2010, TIN2006-15662-C02-02, and TIN2007-68005-C04-04.

[1] $\bar{l}_1 \vee l_2, \bar{l}_1 \vee l_3, \bar{l}_2 \vee \bar{l}_3$ is equivalent to $l_1 \rightarrow l_2, l_1 \rightarrow l_3, \bar{l}_2 \vee \bar{l}_3$.

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 467–480, 2009.

(note that a cycle structure implies a failed literal l_1), and (ii) replace the cycle structure with $\bar{l}_1, l_1 \vee \bar{l}_2 \vee \bar{l}_3, \bar{l}_1 \vee l_2 \vee l_3$ by applying MaxSAT resolution [2,5], which amounts to activate the inference component and, at the same time, change the behaviour of the underestimation component. We first show that it is better to apply MaxSAT resolution to cycle structures occurring in inconsistent subformulas detected using unit propagation or failed literal detection. We then propose a heuristic that guides the application of MaxSAT resolution to cycle structures during failed literal detection, and evaluate this heuristic by implementing it in MaxSatz, obtaining a new solver called MaxSatz$_c$. Our experimental investigation on weighted MaxSAT and Partial MaxSAT instances shows that MaxSatz$_c$ substantially improves MaxSatz on many hard random, crafted and industrial instances.

This paper extends the results of [7] to Weighted MaxSAT and Partial MaxSAT, and includes experiments with random, crafted and industrial instances of the last MaxSAT Evaluation (in [7], the results are only for unweighted MaxSAT, and experiments are limited to Max-2SAT instances). The implementations in [7] were performed on top of an optimized version of MaxSatz for unweighed MaxSAT that was first used in the 2007 MaxSAT Evaluation, but the implementations of this paper were performed on top of an optimized version of MaxSatz for Weighted Partial MaxSAT that was used in the 2008 MaxSAT Evaluation. This paper also contains two new lemmas (Lemma 1 and Lemma 2), a formal proof of Proposition 1, an example (Example 2) that shows that applying MaxSAT resolution to cycle structures not contained in an inconsistent subformula may lead to worse LBs, and a deeper analysis of the experimental results. For the sake of clarity, we explain our work for unweighted MaxSAT, but the implementation and experiments include Weighted MaxSAT and Partial MaxSAT.

2 Preliminaries

We define CNF formulas as multisets of clauses because, in Max-SAT, duplicated clauses cannot be collapsed into one clause, and define weighted CNF formulas as multisets of weighted clauses. A weighted clause is a pair (C_i, w_i), where C_i is a disjunction of literals and w_i, its weight, is a positive number. The weighted clauses $(C, w_i), (C, w_j)$ can be replaced with $(C, w_i + w_j)$. A literal l in a (weighted) CNF formula ϕ is *failed* if unit propagation derives a contradiction from $\phi \wedge l$ but not from ϕ. An empty clause cannot be satisfied and is denoted by $\square$.

The *(Unweighted) MaxSAT problem* for a CNF formula ϕ is the problem of finding a truth assignment that maximizes (minimizes) the number of satisfied (unsatisfied) clauses.[2] MaxSAT instances ϕ_1 and ϕ_2 are equivalent if ϕ_1 and ϕ_2 have the same number of unsatisfied clauses for every complete assignment of ϕ_1 and ϕ_2. A MaxSAT inference rule is sound if it transforms an instance into an equivalent instance.

The *Weighted MaxSAT problem* for a weighted CNF formula ϕ is the problem of finding an assignment that minimizes the sum of weights of unsatisfied clauses. A *Partial MaxSAT* instance is a CNF formula in which some clauses are *relaxable* or *soft* and the rest are *non-relaxable* or *hard*. Solving a Partial MaxSAT instance amounts to find an assignment that satisfies all the hard clauses and the maximum number of soft clauses.

[2] In the sequel, we always refer to the minimization version of MaxSAT, also called MinUNSAT.

3 Related Work

3.1 Underestimation Component

The underestimation in LB UP [8] is the number of disjoint inconsistent subformulas that can be detected with unit propagation. UP works as follows: it applies unit propagation until a contradiction is derived. Then, UP identifies, by inspecting the implication graph created by unit propagation, a subset of clauses from which a unit refutation can be constructed, and tries to derive new contradictions from the remaining clauses. The order in which unit clauses are propagated has a clear impact on the quality of the LB [9]. Recently, Darras et al. [3] and Han et al. [11] have developed two versions of UP in which the computation of the LB is made more incremental.

UP can be enhanced with failed literal detection (UP_{FL}) [9] : Given a MaxSAT instance ϕ *to which we have already applied UP*, and a variable x, UP_{FL} applies UP to $\phi \wedge x$ and $\phi \wedge \bar{x}$. If UP derives a contradiction from both $\phi \wedge x$ and $\phi \wedge \bar{x}$, then the union of the two inconsistent subformulas identified by UP respectively in $\phi \wedge x$ and $\phi \wedge \bar{x}$ is an inconsistent subformula of ϕ, after excluding x and $\bar{x}$. Since applying failed literal detection to every variable is time consuming, it is only applied to the variables which do not occur in unit clauses, and have at least two positive and two negative occurrences in binary clauses. Once an inconsistent subformula γ is detected, γ is removed from ϕ, the underestimation is increased by one, and UP_{FL} continues in the modified ϕ.

In this paper, when we say an inconsistent subformula, we mean an inconsistent subformula detected using unit propagation or failed literal detection.

Another approach for computing underestimations is based on first reducing the MaxSAT instance one wants to solve to an instance of another problem, and then solving a relaxation of the obtained instance. For example, Clone [12] and SR(w) [13] solve the minimum cardinality problem of a deterministic decomposable negation normal form (d-DNNF) compilation of a relaxation of the current MaxSAT instance.

3.2 Inference Component

An alternative to improve the quality of the LB consists in applying MaxSAT resolution. In practice, competitive solvers apply some refinements of the rule for efficiency reasons. In contrast to SAT resolution, a MaxSAT inference rule replaces the clauses in the premises with the clauses in the conclusion in order to preserve the number of unsatisfied clauses. If the conclusion would be added to the premises as in SAT resolution, the number of unsatisfied clauses might increase.

MaxSatz [10] incorporates the following rules (also called Rule 1, Rule 2, Rule 3, Rule 4 in this paper) capturing special structures in a MaxSAT instance:

$$l_1,\ \bar{l}_1 \vee \bar{l}_2,\ l_2 \implies \square,\ l_1 \vee l_2 \tag{1}$$

$$l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_2 \vee l_3, \cdots,\ \bar{l}_k \vee l_{k+1},\ \bar{l}_{k+1} \implies \square,\ l_1 \vee \bar{l}_2,\ l_2 \vee \bar{l}_3, \cdots,\ l_k \vee \bar{l}_{k+1} \tag{2}$$

$$l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_1 \vee l_3,\ \bar{l}_2 \vee \bar{l}_3 \implies \square,\ l_1 \vee \bar{l}_2 \vee \bar{l}_3,\ \bar{l}_1 \vee l_2 \vee l_3 \tag{3}$$

$$\begin{array}{c} l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_2 \vee l_3, \cdots,\ \bar{l}_k \vee l_{k+1}, \\ \bar{l}_{k+1} \vee l_{k+2},\ \bar{l}_{k+1} \vee l_{k+3},\ \bar{l}_{k+2} \vee \bar{l}_{k+3} \end{array} \Longrightarrow \begin{array}{c} \square,\ l_1 \vee \bar{l}_2,\ l_2 \vee \bar{l}_3, \cdots,\ l_k \vee \bar{l}_{k+1}, \\ l_{k+1} \vee \bar{l}_{k+2} \vee \bar{l}_{k+3},\ \bar{l}_{k+1} \vee l_{k+2} \vee l_{k+3} \end{array} \quad (4)$$

Max-DPLL [6] incorporates several rules for weighted MaxSAT, including chain resolution (which is equivalent to Rule 2 in the unweighted case) and cycle resolution. Cycle resolution, which captures the cycle structure, is implemented for 3 variables:

$$\bar{l}_1 \vee l_2,\ \bar{l}_1 \vee l_3,\ \bar{l}_2 \vee \bar{l}_3 \Longrightarrow \bar{l}_1,\ l_1 \vee \bar{l}_2 \vee \bar{l}_3,\ \bar{l}_1 \vee l_2 \vee l_3 \quad (5)$$

MiniMaxSat incorporates LB UP and, once a contradiction is found, it applies MaxSAT resolution to the detected inconsistent subformula if the largest resolvent in the refutation has arity less than 4; otherwise, it just increments the underestimation.

Max-DPLL applies MaxSAT resolution, via the cycle resolution inference rule, to all the cycle structures occurring in a MaxSAT instance, and does not combine its application with the underestimation component. MaxSatz and MiniMaxSat both select cycle structures to which MaxSAT resolution can be applied. MaxSatz applies MaxSAT resolution, via Rule 3 and Rule 4, just when unit propagation detects a contradiction containing the cycle structure. MiniMaxSat applies MaxSAT resolution to cycles structures which are contained in an inconsistent subformula detected by UP provided that the largest resolvent in the refutation of the subformula has arity less than 4.

4 Cycle Structures and Lower Bounds

Exploiting cycle structures has proved very useful in Max-DPLL, MaxSatz, and MiniMaxSat. In this section, we study why and when exploiting cycle structures is useful.

The operation of Rule 3 and Rule 4 of MaxSatz can be analyzed as follows. Given an inconsistent subformula $\{l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_1 \vee l_3,\ \bar{l}_2 \vee \bar{l}_3\}$ detected using unit propagation, Rule 3 applies MaxSAT resolution to transform the subformula into $\{l_1, \bar{l}_1,\ l_1 \vee \bar{l}_2 \vee \bar{l}_3,\ \bar{l}_1 \vee l_2 \vee l_3\}$, and then into $\{\square,\ l_1 \vee \bar{l}_2 \vee \bar{l}_3,\ \bar{l}_1 \vee l_2 \vee l_3\}$ (since $\{l_1, \bar{l}_1\}$ is equivalent to $\square$). The benefit of the transformation is twofold: (i) the empty clause does not need to be re-detected in the subtree rooted at the current node because it remains in the transformed formula, and (ii) the transformed subformula includes two new ternary clauses, and such *liberated* clauses may be used to detect further inconsistent subformulas, allowing to compute better LBs. The case of Rule 4 is similar.

The next example illustrates the usefulness of applying MaxSAT resolution to cycle structures in scenarios where there is no unit clause.

Example 1. Assume that a MaxSAT instance ϕ contains

$$\begin{array}{l} x_1 \vee x_2,\ \bar{x}_2 \vee x_3,\ \bar{x}_2 \vee x_4,\ \bar{x}_3 \vee \bar{x}_4,\ \bar{x}_1 \vee x_5,\ \bar{x}_5 \vee x_6,\ \bar{x}_1 \vee x_7,\ \bar{x}_6 \vee \bar{x}_7 \\ x_8 \vee \bar{x}_2,\ x_8 \vee x_3,\ x_8 \vee x_4,\ \bar{x}_8 \vee x_9,\ \bar{x}_8 \vee x_{10},\ \bar{x}_8 \vee x_{11},\ \bar{x}_9 \vee \bar{x}_{10} \vee \bar{x}_{11} \end{array}$$

Rule 3 and Rule 4 are not applied since there is no unit clause. Failed literal detection on the variable x_1 finds the inconsistent subformula in the first line. After removing this

subformula, it cannot detect further inconsistent subformulas. The underestimation is only incremented by 1. However, if MaxSAT resolution is applied to

$$\bar{x}_2 \vee x_3,\ \bar{x}_2 \vee x_4,\ \bar{x}_3 \vee \bar{x}_4$$

in the first line, these clauses are replaced with

$$\bar{x}_2,\ x_2 \vee \bar{x}_3 \vee \bar{x}_4,\ \bar{x}_2 \vee x_3 \vee x_4$$

and then the underestimation component detects 2 inconsistent subformulas instead of 1. The first with failed literal detection on the variable x_1:

$$x_1 \vee x_2,\ \bar{x}_2,\ \bar{x}_1 \vee x_5,\ \bar{x}_5 \vee x_6,\ \bar{x}_1 \vee x_7,\ \bar{x}_6 \vee \bar{x}_7$$

and the second with failed literal detection on the variable x_8:

$$x_8 \vee \bar{x}_2,\ x_8 \vee x_3,\ x_8 \vee x_4,\ \bar{x}_8 \vee x_9,\ \bar{x}_8 \vee x_{10},\ \bar{x}_8 \vee x_{11},\ \bar{x}_9 \vee \bar{x}_{10} \vee \bar{x}_{11},\ x_2 \vee \bar{x}_3 \vee \bar{x}_4$$

Example 1, together with the analysis of Rule 3 and Rule 4, suggests that one should apply MaxSAT resolution to cycle structures contained in an inconsistent subformula to improve the quality of LBs. In fact, generally speaking, let ϕ be a MaxSAT instance and l a literal of ϕ, we have

Lemma 1. *Let l be a failed literal in ϕ (i.e., UP($\phi \wedge l$) derives an empty clause), and let S_l be the set of clauses used to derive the contradiction in UP($\phi \wedge l$). If S_l contains the cycle structure $\bar{l}_1 \vee l_2,\ \bar{l}_1 \vee l_3,\ \bar{l}_2 \vee \bar{l}_3$, then l_1 was set to true in the unit propagation.*

Proof. Except the empty clause, every clause in S_l becomes unit when it is used for propagation, meaning that every clause in S_l is satisfied by at most one literal in the unit propagation. If l_1 was set to false in the propagation, then at least one of the three clauses $\bar{l}_1 \vee l_2,\ \bar{l}_1 \vee l_3,\ \bar{l}_2 \vee \bar{l}_3$ would be satisfied by two literals and could not belong to S_l. So, l_1 was set to true in the unit propagation. ■

Lemma 2. *If l is a failed literal, and S_l contains the cycle structure $\bar{l}_1 \vee l_2,\ \bar{l}_1 \vee l_3,\ \bar{l}_2 \vee \bar{l}_3$, then l_1 was assigned a truth value before l_2 and l_3 in UP($\phi \wedge l$).*

Proof. Except the empty clause, every clause in S_l was unit when it was satisfied in the unit propagation. We assume that l_2 was assigned a truth value before l_1 and show that this is impossible. If l_2 was assigned true, clause $\bar{l}_1 \vee l_2$ would be satisfied without being unit; if l_2 was assigned false, then l_3 would be assigned true before l_2 was assigned false, since otherwise clause $\bar{l}_2 \vee \bar{l}_3$ would be satisfied before being unit. But in the latter case, clause $\bar{l}_1 \vee l_3$ would be satisfied without being unit. ■

Lemma 2 also means that if S_l contains a cycle structure, then the cycle structure must be the last three binary clauses in the implication graph detecting S_l, which makes the identification of the cycle structure in S_l fast and easy.

Proposition 1. *Let l be a failed literal in ϕ (i.e., UP($\phi \wedge l$) derives an empty clause), and let S_l be the set of clauses used to derive the contradiction in UP($\phi \wedge l$). If S_l contains the cycle structure $\bar{l}_1 \vee l_2,\ \bar{l}_1 \vee l_3,\ \bar{l}_2 \vee \bar{l}_3$, and S'_l is S_l after applying MaxSAT resolution to the cycle structure, then $S'_l - \{l_1 \vee \bar{l}_2 \vee \bar{l}_3,\ \bar{l}_1 \vee l_2 \vee l_3\}$ is inconsistent.*

Proof. By Lemma 1 and Lemma 2, l_1 was assigned true in UP($\phi \wedge l$) independently of the three clauses $\bar{l}_1 \vee l_2$, $\bar{l}_1 \vee l_3$, $\bar{l}_2 \vee \bar{l}_3$, which are replaced with $\{\bar{l}_1, l_1 \vee \bar{l}_2 \vee \bar{l}_3, \bar{l}_1 \vee l_2 \vee l_3\}$ in S'_l. So, unit propagation in $S'_l - \{l_1 \vee \bar{l}_2 \vee \bar{l}_3, \bar{l}_1 \vee l_2 \vee l_3\}$ derives an empty clause from the unit clause $\bar{l}_1$. ■

Proposition 1 means that, if both l and $\bar{l}$ are failed literals, and S_l contains the cycle structure $\bar{l}_1 \vee l_2$, $\bar{l}_1 \vee l_3$, $\bar{l}_2 \vee \bar{l}_3$, we can apply MaxSAT resolution in ϕ (replacing these three binary clauses with one unit clause $\bar{l}_1$ and two ternary clauses $l_1 \vee \bar{l}_2 \vee \bar{l}_3$ and $\bar{l}_1 \vee l_2 \vee l_3$) to obtain S'_l, and then transform the inconsistent subformula $S'_l \cup S_{\bar{l}} - \{l, \bar{l}\}$ into a smaller inconsistent subformula $S'_l \cup S_{\bar{l}} - \{l, \bar{l},\ l_1 \vee \bar{l}_2 \vee \bar{l}_3, \bar{l}_1 \vee l_2 \vee l_3\}$ of ϕ. So, apart from incrementing the underestimation by 1, this transformation liberates two ternary clauses from $S'_l \cup S_{\bar{l}} - \{l, \bar{l}\}$ that can be used to derive other disjoint inconsistent subformulas, allowing to obtain better LBs.

In Example 1, $S_{\bar{x}_1} \cup S_{x_1} - \{\bar{x}_1, x_1\}$ is equal to

$$\{x_1 \vee x_2,\ \bar{x}_2 \vee x_3,\ \bar{x}_2 \vee x_4,\ \bar{x}_3 \vee \bar{x}_4,\ \bar{x}_1 \vee x_5,\ \bar{x}_5 \vee x_6,\ \bar{x}_1 \vee x_7,\ \bar{x}_6 \vee \bar{x}_7\}$$

which is transformed by applying MaxSAT resolution to the cycle structure $\bar{x}_2 \vee x_3$, $\bar{x}_2 \vee x_4$, $\bar{x}_3 \vee \bar{x}_4$ into a smaller inconsistent subformula

$$\{x_1 \vee x_2,\ \bar{x}_2,\ \bar{x}_1 \vee x_5,\ \bar{x}_5 \vee x_6,\ \bar{x}_1 \vee x_7,\ \bar{x}_6 \vee \bar{x}_7\}$$

So, after incrementing the underestimation by 1, we have two additional ternary clauses ($x_2 \vee \bar{x}_3 \vee \bar{x}_4$ and $\bar{x}_2 \vee x_3 \vee x_4$) liberated from the cycle structure which can be used to detect further inconsistent subformulas.

The benefit of applying MaxSAT resolution to a cycle structure not contained in an inconsistent subformula is not so clear. The next example suggests that this application may lead to a worse LB.

Example 2. Assume that a MaxSAT instance ϕ contains

$$\begin{array}{l} x_1 \vee \bar{x}_2,\ x_1 \vee \bar{x}_3,\ x_2 \vee x_3,\ x_2 \vee x_6,\ x_3 \vee \bar{x}_6, \\ \bar{x}_1 \vee x_7,\ \bar{x}_1 \vee x_8,\ \bar{x}_7 \vee x_9,\ \bar{x}_8 \vee \bar{x}_9, \\ \bar{x}_4,\ x_4 \vee \bar{x}_2,\ \bar{x}_3 \vee x_5,\ \bar{x}_5 \end{array}$$

Without activating the inference component, unit propagation detects an inconsistent subformula

$$\{x_2 \vee x_3,\ \bar{x}_4,\ x_4 \vee \bar{x}_2,\ \bar{x}_3 \vee x_5,\ \bar{x}_5\}$$

Then, after removing this subformula from ϕ, failed literal detection on x_1 (i.e., unit propagation in $\phi \wedge \bar{x}_1$ and in $\phi \wedge x_1$ respectively) finds the second inconsistent subformula

$$\{x_1 \vee \bar{x}_2,\ x_1 \vee \bar{x}_3,\ x_2 \vee x_6,\ x_3 \vee \bar{x}_6,\ \bar{x}_1 \vee x_7,\ \bar{x}_1 \vee x_8,\ \bar{x}_7 \vee x_9,\ \bar{x}_8 \vee \bar{x}_9\}$$

Note that the first three clauses of ϕ form a cycle structure but do not belong to a same inconsistent subformula detected using unit propagation or failed literal detection. If MaxSAT resolution is applied to the cycle structure, ϕ becomes

$$\begin{array}{l} x_1,\ \ \bar{x}_1 \vee x_2 \vee x_3,\ x_1 \vee \bar{x}_2 \vee \bar{x}_3,\ x_2 \vee x_6,\ x_3 \vee \bar{x}_6, \\ \bar{x}_1 \vee x_7,\ \bar{x}_1 \vee x_8,\ \bar{x}_7 \vee x_9,\ \bar{x}_8 \vee \bar{x}_9, \\ \bar{x}_4,\ x_4 \vee \bar{x}_2,\ \bar{x}_3 \vee x_5,\ \bar{x}_5. \end{array}$$

Once unit propagation detects the inconsistent subformula

$$\{x_1,\ \bar{x}_1 \vee x_2 \vee x_3,\ \bar{x}_4,\ x_4 \vee \bar{x}_2,\ \bar{x}_3 \vee x_5,\ \bar{x}_5\}$$

ϕ becomes (after removing the inconsistent subformula)

$$\{x_1 \vee \bar{x}_2 \vee \bar{x}_3,\ x_2 \vee x_6,\ x_3 \vee \bar{x}_6,\ \bar{x}_1 \vee x_7,\ \bar{x}_1 \vee x_8,\ \bar{x}_7 \vee x_9,\ \bar{x}_8 \vee \bar{x}_9\}$$

and is consistent. So, only one inconsistent subformula is detected when MaxSAT resolution is applied to the cycle structure, making the LB worse.

5 Heuristics for Applying MaxSAT Resolution in Cycle Structures

From the previous analysis, we observe that it is better to apply MaxSAT resolution to cycle structures contained in an inconsistent subformula in order to transform the inconsistent subformula and liberate two ternary clauses from the subformula. In practice, when we identify a cycle structure at a node of the search tree, we distinguish three cases:

1. The cycle structure is contained in an inconsistent subformula.
2. The cycle structure is not contained in an inconsistent subformula at the current node, but probably belongs to an inconsistent subformula in the subtree below the current node.
3. The cycle structure is not contained in an inconsistent subformula at the current node and probably will not belong to an inconsistent subformula in the subtree.

We define a heuristic that applies MaxSAT resolution in the first two cases. As we will see, the benefit of applying MaxSAT resolution in the second case is twofold: two ternary clauses are liberated in advance, and the probable inconsistent subformula containing the cycle structure will be easier and faster to detect in the subtree with the application of MaxSAT resolution. This heuristic is implemented in Algorithm 1, where occ2(l) is the number of occurrences of literal l in the binary clauses of ϕ.

Between the two literals of a variable x that have reasonable probability to be failed (since their satisfaction results in at least two new unit clauses), Algorithm 1 detects first the literal l with more occurrences in binary clauses. Note that l has a smaller probability of being failed than $\bar{l}$ since its satisfaction produces fewer new unit clauses than the satisfaction of $\bar{l}$.

If l is a failed literal and S_l contains a cycle structure, the cycle structure is replaced to obtain S'_l before detecting $\bar{l}$. If $\bar{l}$ also is a failed literal, the inconsistent subformula $S'_l \cup S_{\bar{l}} - \{l, \bar{l}\}$ is transformed into a smaller inconsistent subformula to liberate two ternary clauses. If $\bar{l}$ is not a failed literal in the current node, failed literal detection does not detect an inconsistent subformula containing the cycle structure of S_l, but S_l is now smaller thanks to MaxSAT resolution because it becomes now $S'_l - \{l_1 \vee \bar{l}_2 \vee \bar{l}_3, \bar{l}_1 \vee l_2 \vee l_3\}$ by Proposition 1, and it will be easier to re-detect in the subtree. Note that $\bar{l}$ has reasonable probability to be a failed literal in the subtree, i.e., the cycle structure in the original S_l has reasonable probability to be contained in an inconsistent subformula in the subtree. As soon as $\bar{l}$ fails in the subtree, Algorithm 1 will detect the smaller inconsistent subformula $S_l \cup S_{\bar{l}} - \{l, \bar{l}\}$ with smaller cost (recall S_l is now smaller).

Algorithm 1. *flAndCycle*(ϕ, x), combining MaxSAT resolution to cycle structures and failed literal detection

Input: A MaxSAT instance ϕ, and a variable x such that occ2(x)$\geq$ 2 and occ2($\bar{x}$)$\geq$ 2
Output: ϕ in which MaxSAT resolution is possibly applied to a cycle structure, and an underestimation

```
1 begin
2   if occ2(x)>occ2(x̄) then l⟵x; else l⟵x̄;
    underestimation ⟵ 0;
    if UP(φ ∧ l) derives a contradiction then
3     if S_l contains a cycle structure, replace the cycle structure with one unit clause and two ternary clauses;
      if UP(φ ∧ l̄) derives a contradiction then
4       if S_l̄ contains a cycle structure, replace the cycle structure with one unit clause and two ternary clauses;
        underestimation ⟵ 1;
5   return new φ and underestimation
6 end
```

On the contrary, if l is not a failed literal, Algorithm 1. does not detect an inconsistent subformula, $\bar{l}$ is not detected and no inference is applied to $S_{\bar{l}}$ even if $\bar{l}$ is a failed literal, avoiding the application of MaxSAT resolution to a cycle structure not contained in an inconsistent subformula.

With the aim of evaluating the impact of Algorithm 1. in the performance of MaxSatz, we define the following variants of solvers:

- MaxSatz: It is a Weighted Partial MaxSAT solver developed by J. Argelich, C.M. Li and F. Manyà [1]. MaxSatz participated in the 2008 MaxSAT Evaluation, and incorporates all the MaxSatz inference rules, and failed literal detection, besides UP, in the underestimation component. MaxSatz applies MaxSAT resolution to cycles structures in a limited way using Rule 3 and Rule 4.
- MaxSatz$_c$: It is a variant of MaxSatz in which failed literal detection is combined with the heuristic application of MaxSAT resolution to cycle structures. For every variable x such that occ2(x)$\geq$ 2 and occ2($\bar{x}$)$\geq$ 2, failed literal detection is replaced with Algorithm 1. For the rest of variables, it is applied as in MaxSatz.
- MaxSatz$_c^p$: It is a variant of MaxSatz$_c$ in which MaxSAT resolution is applied to all the cycle structures appearing at the root node, and is applied as in MaxSatz$_c$ to the cycle structures appearing in the rest of nodes. In other words, MaxSAT resolution is exhaustively applied to cycle structures as a preprocessing. Notice that this preprocessing has no effect on problems not containing cycle structures in the input formula (e.g., Max-3SAT). In this case, MaxSatz$_c^p$ is just MaxSatz$_c$. Although all cycle structures at the root node are replaced, new cycle structures can be created in the rest of nodes. Cycle structures may appear because (i) non-binary clauses may become binary clauses during the search, and (ii) Rule 1, Rule 2, Rule 3, and Rule 4 applied in UP, before failed literal detection, may transform binary clauses and add ternary clauses.

- MaxSatz^p: It is a variant of MaxSatz in which MaxSAT resolution is applied to all the cycle structures appearing at the root node, and in the rest of nodes, it is just MaxSatz.
- MaxSatz$_{c^*}$: It is a variant of MaxSatz in which MaxSAT resolution is applied exhaustively to all cycle structures at each node after applying UP and inference rules (Rule 1, Rule 2, Rule 3, and Rule 4), and before applying failed literal detection. The application is exhaustive because no subset of binary clauses matching a cycle structure remains in the current instance.

MaxSatz$_{c^*}$ is related to Max-DPLL when replacing every cycle structure with one unit clause and two ternary clauses. MaxSatz$_c$ extends MaxSatz and MiniMaxSat in that MaxSatz$_c$ additionally applies MaxSAT resolution to cycle structures contained in an inconsistent subformula detected using failed literal detection. Moreover, differently from MiniMaxSat, MaxSatz$_c$ replaces these cycle structures no matter if the refutation has arity less than 4 or not.

Recall that a weighted clause (C, w_1+w_2) is equivalent to two weighted clauses (C, w_1) and (C, w_2). The difference between MaxSatz$_c$ and MaxSatz$_{c^*}$ should be bigger for Weighted MaxSAT than for unweighted MaxSAT. For example, if a weighted formula contains a cycle structure ($\bar{l}_1 \vee l_2$, 3), ($\bar{l}_1 \vee l_3$, 4), ($\bar{l}_2 \vee \bar{l}_3$, 5), MaxSatz$_{c^*}$ replaces entirely this cycle structure with ($\bar{l}_1$, 3), ($l_1 \vee \bar{l}_2 \vee \bar{l}_3$, 3), ($\bar{l}_1 \vee l_2 \vee l_3$, 3), and leaves two clauses ($\bar{l}_1 \vee l_3$, 1), ($\bar{l}_2 \vee \bar{l}_3$, 2) in the formula. On the contrary, MaxSatz$_c$ only replaces the part of this cycle structure contained in an inconsistent subformula. If the minimum clause weight in the inconsistent subformula is 2, MaxSatz$_c$ replaces this cycle structure with ($\bar{l}_1$, 2), ($l_1 \vee \bar{l}_2 \vee \bar{l}_3$, 2), ($\bar{l}_1 \vee l_2 \vee l_3$, 2), and leaves the cycle structure ($\bar{l}_1 \vee l_2$, 1), ($\bar{l}_1 \vee l_3$, 2), ($\bar{l}_2 \vee \bar{l}_3$, 3) in the formula, which is different from the unweighted case where MaxSatz$_c$ *never partly* replaces a cycle structure.

6 Experimental Results and Analysis

We conducted experiments to compare the performance of the different versions of MaxSatz described in the previous section (MaxSatz, MaxSatz$_c$, MaxSatz$_c^p$, MaxSatz^p, and MaxSatz$_{c^*}$).

As benchmarks for Weighted MaxSAT, we considered random weighted Max-2SAT, random weighted Max-3SAT, and random weighted Max-CUT instances, and all the crafted instances of the 2008 MaxSAT Evaluation (the evaluation did not include industrial instances for Weighted MaxSAT). As benchmarks for Partial MaxSAT, we considered random partial Max-2SAT and random partial Max-3SAT instances, and all the industrial and crafted instances of the 2008 MaxSAT Evaluation. We did not solve the random instances of the Weighted MaxSAT and Partial MaxSAT categories of the 2008 MaxSAT Evaluation because they are easily solved with the different versions of MaxSatz. We selected instances which are harder and allow to analyze the scaling behavior of the solvers.

We do not include the experimental results with other solvers in this section for three reasons: (i) the purpose of the experiments is to show the effectiveness of the heuristic replacement of cycles structures given in Algorithm 1, while keeping other things equal

in a solver; (ii) for randomly generated (weighted or partial) instances, other solvers are too slow to be displayed in the figures; (iii) for crafted and industrial instances of the 2008 MaxSAT Evaluation, the performance of other solvers can be checked in the web page of the evaluation (http//www.maxsat.udl.cat/08/).

Experiments were performed on a MacPro with two 2.8GHz Quad-Core Intel Xeon processors and 4Gb of RAM. For every instance, besides the run time, we compute the total number k of cycle structures replaced by applying MaxSAT resolution (in addition to Rule 3 and Rule 4), and divide k by the search tree size t. The ratio k/t roughly indicates the average number of cycle structures replaced (in addition to the applications of Rule 3 and Rule 4) at a search tree node. For the 2008 MaxSAT Evaluation instances, we set a cutoff of 30 minutes. These instances generally include no or very few cycle structures in their initial formulas, so that the preprocessing is not significant. We do not include MaxSatz$_c^p$ and MaxSatz^p in the comparison for them for the sake of clarity.

The experimental results for Weighted MaxSAT are shown in Figure 1, Figure 2, Figure 3, and Table 1. Figure 1 shows the mean time (left plot) and the mean ratio k/t (right plot) to solve sets of 100 randomly generated Weighted Max-2SAT instances with 100 variables and an increasing number of clauses. MaxSatz is not included in the right plot, because cycle structures replaced by Rule 3 and Rule 4 are not counted in k, so that k is always 0 for MaxSatz. We observe a clear advantage of MaxSatz$_c$ and MaxSatz$_c^p$, which are up to one order of magnitude better than MaxSatz and MaxSatz$_{c^*}$. The search tree size (not shown for lack of space) follows the same ordering. It can be observed that, it suffices to replace three or four cycle structures contained in an inconsistent subformula at a search tree node to obtain an important gain, and the gain grows with the number of cycle structures replaced. However, if the cycle structures not contained in an inconsistent subformula are also replaced as in MaxSatz$_{c^*}$, the LB becomes substantially worse and the search tree larger, and the more replacements there are, the worse the LB.

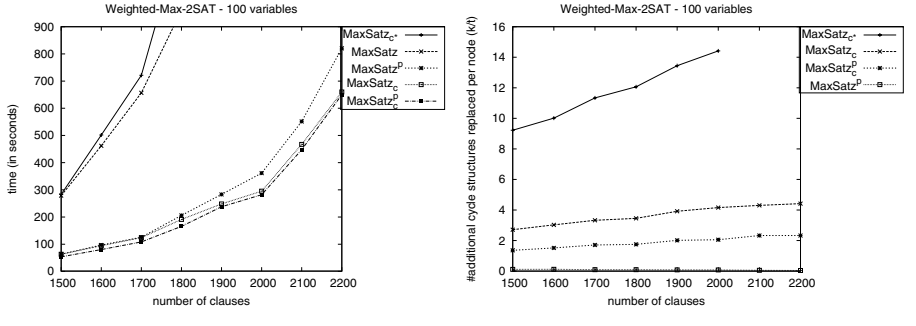


Fig. 1. Weighted Max-2SAT instances

Figure 2 shows the mean time (left plot) and the mean ratio k/t (right plot) to solve sets of 100 randomly generated Weighted Max-3SAT instance with 60 variables and an increasing number of clauses. Since weighted Max-3SAT instances do not include cycle structures, the preprocessing has no effect. Therefore, Figure 2 does not include

MaxSatz$_c^p$ and MaxSatz^p. We observe that MaxSatz$_c$ is clearly better than the rest of solvers. Note that all cycle structures are dynamically created during search.

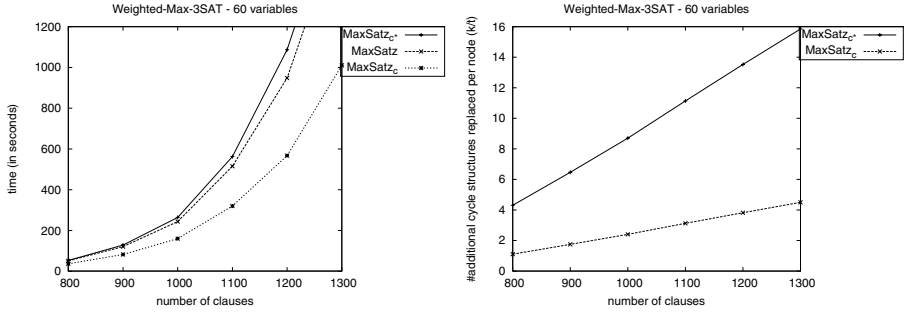


Fig. 2. Weighted Max-3SAT instances

Figure 3 shows the mean time (left plot) and the mean ratio k/t (right plot) to solve sets of 100 random Weighted Max-CUT instances generated from random graphs of 100 nodes and an increasing number of edges. MaxSatz$_{c^*}$ is better than MaxSatz because a cycle structure easily belongs to an inconsistent subformula due to the special structure of the Max-CUT problem. Nevertheless the heuristic application of MaxSAT resolution to cycle structures contained in an inconsistent subformula makes MaxSatz$_c$ significantly better than MaxSatz$_{c^*}$.

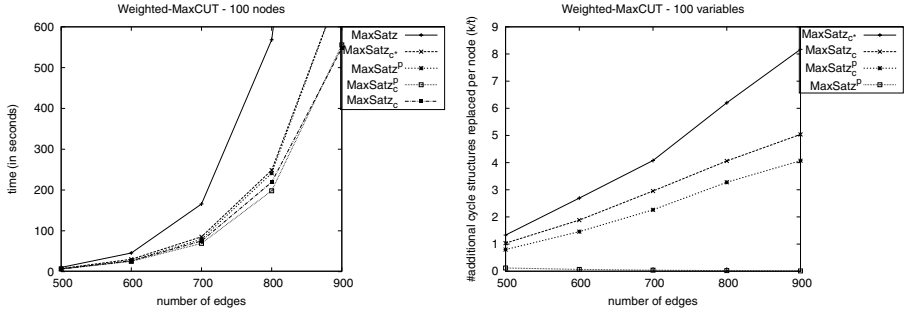


Fig. 3. Weighted Max-CUT instances

Table 1 contains the results for the crafted instances of the Weighted Max-SAT category of the 2008 MaxSAT Evaluation. For each group of instances, we display the number of instances **I** in the group, and for each solver, the number of instances solved within the cutoff of 30 minutes (in brackets) and the mean time in seconds to solve these solved instances. MaxSatz$_c$ is the best performing solver, which solves 4 instances more than MaxSatz and 2 instances more than Maxsatz$_{c^*}$.

Table 1. Crafted instances of the Weighted Max-SAT category of the 2008 MaxSAT Evaluation

Instance set	I	MaxSatz	MaxSatz$_c$	MaxSatz$_{c*}$
KeXu	15	14.97(10)	**13.85(10)**	25.28(10)
RAMSEY	48	25.45(36)	**10.93(36)**	14.80(36)
WMAXCUT-DIMACS-MOD	62	54.22(55)	**90.95(57)**	89.34(55)
WMAXCUT-RANDOM	40	10.22(40)	**3.54(40)**	5.38(40)
WMAXCUT-SPINGLASS	5	301.83(2)	**31.23(4)**	35.60(4)
All instances	170	143	**147**	145

The experimental results for Partial MaxSAT are shown in Figure 4, Figure 5, Table 2, and Table 3. Figure 4 shows the mean time (left plot) and the mean ratio k/t (right plot) to solve sets of 100 randomly generated partial Max-2SAT instance with 150 variables, 150 hard clauses as in the 2008 MaxSAT evaluation, and an increasing number of soft clauses. For these large instances, MaxSatz$_c$ outperforms the rest of solvers, and MaxSatz$_{c*}$ is by far the worst.

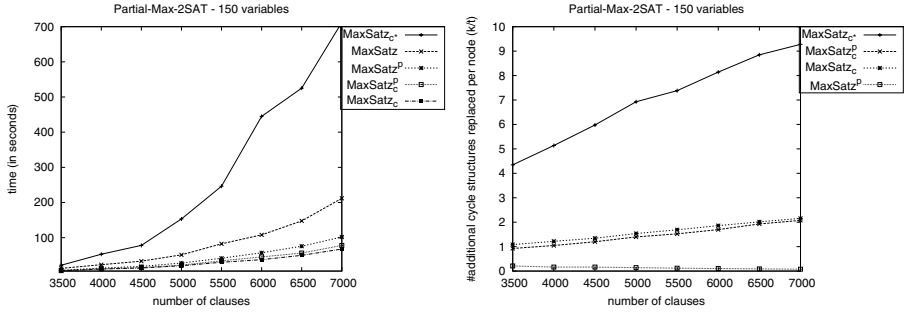


Fig. 4. Partial Max-2SAT instances

Figure 5 shows the mean time (left plot) and the mean ratio k/t (right plot) to solve sets of 100 randomly generated partial Max-3SAT instance with 80 variables, 80 hard clauses, and an increasing number of soft clauses. Note that there are very few cycle structures replaced at a search tree node, but the gain of MaxSatz$_c$ and the loss of MaxSatz$_{c*}$ are very significant.

Table 2 contains the results for the industrial instances of the Partial Max-SAT category of the 2008 MaxSAT Evaluation. MaxSatz$_c$ solves 25 instances more than MaxSatz, and 40 instances more than MaxSatz$_{c*}$. Table 3 contains the results for the crafted instances of the Partial Max-SAT category of the 2008 MaxSAT Evaluation. MaxSatz$_c$ solves 7 instances more than MaxSatz$_{c*}$. There are very few cycle structures contained in an inconsistent subformula during search for these instances. Their exploitation still makes MaxSatz$_c$ the best performing solver in general.

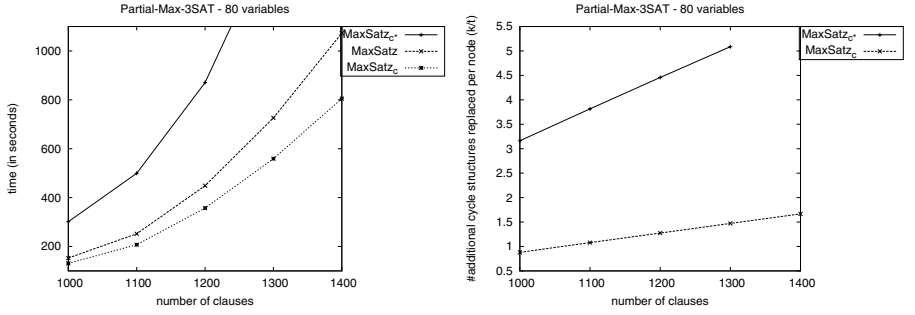


Fig. 5. Partial Max-3SAT instances

Table 2. Industrial instances of the Partial Max-SAT category of the 2008 MaxSAT Evaluation

Instance set	I	MaxSatz	MaxSatz$_c$	MaxSatz$_{c^*}$
bcp-fir	59	8.91(7)	**5.03(7)**	8.16(7)
bcp-hipp-yRa1	1183	73.18(734)	**70.81(744)**	77.01(721)
bcp-msp	148	40.53(94)	**17.86(94)**	22.41(94)
bcp-mtg	215	33.07(144)	**96.25(157)**	95.41(154)
bcp-syn	74	**97.41(22)**	104.68(22)	82.67(21)
pbo-mqc-nencdr	128	**514.05(77)**	436.91(76)	475.17(64)
pbo-mqc-nlogencdr	128	323.57(104)	**270.38(107)**	333.04(106)
pbo-routing	15	61.43(5)	**3.13(5)**	5.22(5)
All instances	1950	1187	**1212**	1172

Table 3. Crafted instances of the Partial Max-SAT category of the 2008 MaxSAT Evaluation

Instance set	I	MaxSatz	MaxSatz$_c$	MaxSatz$_{c^*}$
MAXCLIQUE-RANDOM	96	75.67(83)	**68.17(83)**	48.68(80)
MAXCLIQUE-STRUCTURED	62	164.70(25)	**138.90(25)**	149.25(22)
MAXONE-3SAT	80	**139.09(78)**	148.70(78)	204.47(77)
MAXONE-STRUCTURED	60	80.63(58)	**75.36(58)**	96.41(58)
All instances	298	244	**244**	237

7 Conclusions

We have studied why and when is useful to apply MaxSAT resolution to cycle structures in MaxSAT LB computation. We found that the exhaustive application of MaxSAT resolution is not so effective in general, and that MaxSAT resolution is effective if it is applied to cycle structures contained in an inconsistent subformula detected using unit propagation or failed literal detection. The benefit is twofold: (i) the inconsistent subformula can be transformed into a smaller one to liberate two ternary clauses for detecting other inconsistent subformulas, (ii) the smaller inconsistent subformula is easier and faster to detect or re-detect in subtrees. Experimental results suggest that the solver becomes

much slower when MaxSAT resolution is applied to cycle structures not contained in an inconsistent subformula.

We defined a heuristic that guides the applications of MaxSAT resolution to cycle structures. The implementation of this heuristic provides empirical evidence that it is very effective on many hard instances of Weighted MaxSAT and Partial MaxSAT, independently if they are random, crafted or industrial, as soon as few inconsistent subformulas contain a cycle structure.

In the future, we will study the exploitation of other structures in MaxSAT instances. It is remarkable that a relevant exploitation of few structures at a search tree node can result in a substantial speed-up in the solving of a MaxSAT problem.

References

1. Argelich, J., Li, C.M., Manyà, F.: An improved exact solver for partial Max-SAT. In: NCP 2007, pp. 230–231 (2007)
2. Bonet, M.L., Levy, J., Manyà, F.: Resolution for Max-SAT. Artificial Intelligence 171(8-9), 240–251 (2007)
3. Darras, S., Dequen, G., Devendeville, L., Li, C.M.: On inconsistent clause-subsets for max-SAT solving. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 225–240. Springer, Heidelberg (2007)
4. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSat: A new weighted Max-SAT solver. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 41–55. Springer, Heidelberg (2007)
5. Larrosa, J., Heras, F.: Resolution in Max-SAT and its relation to local consistency in weighted CSPs. In: IJCAI 2005, pp. 193–198 (2005)
6. Larrosa, J., Heras, F., de Givry, S.: A logical approach to efficient Max-SAT solving. Artificial Intelligence 172(2-3), 204–233 (2008)
7. Li, C.M., Manyà, F., Mohamedou, N.O., Planes, J.: Transforming inconsistent subformulas in MaxSAT lower bound computation. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 582–587. Springer, Heidelberg (2008)
8. Li, C.M., Manyà, F., Planes, J.: Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 403–414. Springer, Heidelberg (2005)
9. Li, C.M., Manyà, F., Planes, J.: Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In: AAAI 2006, pp. 86–91 (2006)
10. Li, C.M., Manyà, F., Planes, J.: New inference rules for Max-SAT. Journal of Artificial Intelligence Research 30, 321–359 (2007)
11. Lin, H., Su, K., Li, C.M.: Within-problem learning for efficient lower bound computation in Max-SAT solving. In: AAAI 2008, pp. 351–356 (2008)
12. Pipatsrisawat, K., Darwiche, A.: Clone: Solving weighted Max-SAT in a reduced search space. In: AI 2007, pp. 223–233 (2007)
13. Ramírez, M., Geffner, H.: Structural relaxations by variable renaming and their compilation for solving MinCostSAT. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 605–619. Springer, Heidelberg (2007)

Generalizing Core-Guided Max-SAT

Mark H. Liffiton and Karem A. Sakallah

Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor 48109-2121
{liffiton,karem}@eecs.umich.edu

Abstract. Recent work has shown the value of using unsatisfiable cores to guide maximum satisfiability algorithms (Max-SAT) running on industrial instances [5,9,10,11]. We take this concept and generalize it, applying it to the problem of finding minimal correction sets (MCSes), which themselves are generalizations of Max-SAT solutions. With the technique's success in Max-SAT for industrial instances, our development of a generalized approach is motivated by the industrial applications of MCSes in circuit debugging [12] and as a precursor to computing minimal unsatisfiable subsets (MUSes) in a hardware verification system [1]. Though the application of the technique to finding MCSes is straightforward, its correctness and completeness are not, and we prove both for our algorithm. Our empirical results show that our modified MCS algorithm performs better, often by orders of magnitude, than the existing algorithm, even in cases where the use of cores has a negative impact on the performance of Max-SAT algorithms.

1 Introduction

In the field of constraint processing, and particularly within the domain of Boolean Satisfiability (SAT), the analysis of infeasible constraint systems has become increasingly important. Following the impressive advancements in the performance of SAT solvers in the past decade, which enable fast answers about the satisfiability of industrially relevant instances, researchers have begun to look at analyses beyond the "unsatisfiable" result returned for overconstrained instances. The work is spurred not only by academic interest but also by novel industrial applications of these analyses. In this paper, we take one of the recent advances in analyzing infeasible instances, namely unsatisfiable-core-guided maximum satisfiability (core-guided Max-SAT), and generalize it to solve a related analysis with direct industrial applications: the identification of minimal correction sets (MCSes).

The concept of core-guided Max-SAT was first developed by Fu & Malik [5] and later enhanced and optimized by Marques-Silva and others [9,10,11]; the algorithms and differences in their approaches are detailed in Section 3. The technique relies on and exploits one of the relationships between satisfiable and unsatisfiable subsets of infeasible systems that have been explored in depth in [3] and [6,7]. Briefly, an unsatisfiable instance will contain one or more *unsatisfiable*

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 481–494, 2009.

cores. No satisfiable subset of such an instance can contain any complete cores; therefore, any Max-SAT solution must must leave unsatisfied at least one clause from every core. The core-guided Max-SAT approach thus identifies unsatisfiable cores of an instance and only considers clauses within those cores as potential "removals," limiting the search space dramatically.

The use of unsatisfiable cores in solving Max-SAT yields drastically different performance than other current Max-SAT techniques, which are generally based on branch-and-bound. In the 2008 Max-SAT Evaluation [2], core-guided Max-SAT algorithms performed extremely well in the industrial Max-SAT category (one solving 72 of 112 instances within the timeout, when other approaches solved 0-3 and, in one case, 10 instances within the timeout), while performing among the bottom of the pack on random and crafted instances.

The industrial Max-SAT instances in the Max-SAT Evaluation are in fact produced by the circuit debugging system in [12], in which the desired result is actually MCSes of the CNF instances. In that work, an algorithm simply called MCSes [7] is used as a preprocessing step, identifying approximations of MCSes which are then used to boost a complete SAT-based search. In this work, motivated by the success of core-guided Max-SAT on these instances, we generalize the core-guided Max-SAT approach to apply it to the problem of finding MCSes of CNF instances. Our new algorithm, MCSes-U, is described and its correctness proven in Section 4, and we present experimental results showing its improvement over MCSes in Section 5.

2 Preliminaries

Boolean satisfiability (SAT) is a problem domain involving Boolean formulas in *conjunctive normal form* (CNF). Formally, a CNF formula φ is defined as follows:

$$\varphi = \bigwedge_{i=1\ldots n} C_i \qquad\qquad C_i = \bigvee_{j=1\ldots k_i} a_{ij}$$

where each *literal* a_{ij} is either a positive or negative instance of some Boolean variable (e.g., x_3 or $\neg x_3$, where the domain of x_3 is $\{0, 1\}$), the value k_i is the number of literals in the *clause* C_i (a disjunction of literals), and n is the number of clauses in the formula. In more general terms, each clause is a *constraint* of the constraint system φ. We will often treat CNF formulas as sets of clauses (*clause sets*), so equivalently: $\varphi = \bigcup_{i=1\ldots n} C_i$.

A CNF instance is said to be *satisfiable* if there exists some assignment to its variables that makes the formula evaluate to 1 or TRUE; otherwise, it is *unsatisfiable*. We will use the following unsatisfiable CNF instance φ as an example.

$$\varphi = (a)(\neg a)(\neg a \vee b)(\neg b)$$

Maximum Satisfiability (Max-SAT) is the problem of, given an unsatisfiable CNF formula, identifying a satisfiable subset of its clauses with maximum cardinality. Alternatively, we can say it is the problem of finding an assignment of the

formula's variables that satisfies a maximum cardinality subset of the clauses. The example formula φ has a single Max-SAT solution: $\{(\neg a), (\neg a \vee b), (\neg b)\}$ are satisfied by $a = \mathrm{F}, b = \mathrm{F}$.

Minimal Correction Sets (MCSes) can be understood as generalizations of Max-SAT solutions. Given any Max-SAT solution in the form of a satisfiable subset of clauses, we can look at those clauses left unsatisfied as a *correction set* (CS), because removing them from the formula *corrects* it, making it satisfiable. Due to the maximum cardinality of a Max-SAT solution, its corresponding correction set has minimum cardinality; no smaller correction sets exist. We generalize this to the concept of minim**al** correction sets: An MCS is a correction set such that all of its proper subsets are *not* correction sets. MCSes are minimal, or irreducible, but not necessarily minim**um**. Every Max-SAT solution indicates an MCS, but there can be more MCSes than those that are complements of a Max-SAT solution. The clause (a), not satisfied in φ's Max-SAT solution, is an MCS, as are $\{(\neg a), (\neg a \vee b)\}$ and $\{(\neg a), (\neg b)\}$.

Unsatisfiable Cores / MUSes: Given an unsatisfiable CNF formula, an *unsatisfiable core* of the formula is any subset of its clauses that is unsatisfiable. A *Minimal Unsatisfiable Subset* (MUS) is then an unsatisfiable core that is minimal in the same sense that an MCS is minimal: every proper subset of an MUS is satisfiable. MUSes are thus minimal/irreducible, but, again, not necessarily minim**um**. There are two MUSes in φ: $\{(a), (\neg a)\}$ and $\{(a), (\neg a \vee b), (\neg b)\}$.

Resolution Proofs: In the process of solving unsatisfiable instances, some SAT solvers produce *resolution proofs* (or *resolution refutations*), directed acyclic graphs containing the resolution steps used to prove unsatisfiability. As a solver progresses, it learns new clauses arising from applying resolution to combinations of existing clauses (e.g., $(x_2 \vee x_5) \wedge (\neg x_5 \vee x_7) \rightarrow (x_2 \vee x_7)$), and the "parent" clauses of each new clause can be stored in a graph structure. With this structure, the provenance of any learned clause can be traced back to a subset of the original clauses from which the learned clause can be derived. When a solver "learns" the empty clause, the instance must be unsatisfiable, and tracing the empty clause's parents back to the original clauses identifies an unsatisfiable core of the instance [17] (the identified core must be unsatisfiable because those clauses can be used to derive the empty clause).

AtMost Constraints are a type of counting or cardinality constraint. Given a set of n literals $\{l_1, l_2, \ldots, l_n\}$ and a positive integer k, s.t. $k < n$, an AtMost constraint is defined as

$$\mathrm{AtMost}(\{l_1, l_2, \ldots, l_n\}, k) \equiv \sum_{i=1}^{n} \mathrm{val}(l_i) \leq k$$

where $\mathrm{val}(l_i)$ is 1 if l_i is assigned TRUE and 0 otherwise. This constraint places an upper bound on the number of literals in the set assigned TRUE.

Clause-Selector Variables can be used to augment a CNF formula in such a way that standard SAT solvers can manipulate and, in effect, reason about the formula's clauses without any modification to the solver itself.

Every clause C_i in a CNF formula φ is augmented with a negated clause-selector variable y_i to give $C'_i = (\neg y_i \vee C_i)$ in a new formula φ'. Notice that each C'_i is an implication, $C'_i = (y_i \rightarrow C_i)$. Assigning a particular y_i the value TRUE implies the original clause, essentially enabling it. Conversely, assigning y_i FALSE has the effect of disabling or removing C_i from the set of constraints, as the augmented clause C'_i is satisfied by the assignment to y_i. This change gives a SAT solver the ability to enable and disable constraints as part of its normal search, checking the satisfiability of different subsets of constraints within a single backtracking search tree.

3 Use of Cores in Max-SAT

As described in Section 1, unsatisfiable cores can be used to guide Max-SAT algorithms by limiting the number of clauses they must consider for "removal" or leaving unsatisfied. Researchers have developed a number of algorithms exploiting this, all using the inexpensive resolution proof method for generating unsatisfiable cores.

Fu & Malik first introduced the idea of using unsatisfiable cores to assist in solving Max-SAT in [5]. They described an algorithm based on "diagnosis" that repeatedly finds a core by the resolution proof method, adds clause-selector variables to the clauses in that core, places a one-hot constraint on those clause-selector variables, and searches for a satisfying solution to the modified problem. Essentially, the algorithm identifies a core in each iteration that must be neutralized (by the removal of a clause) in any Max-SAT solution.

Marques-Silva, Planes, and Manquinho [9,10,11] improved upon Fu & Malik's algorithm, which they dubbed MSU1, with several refinements and optimizations. In [10] and [11], Marques-Silva and Planes introduce algorithms MSU1.1, MSU3, and MSU4[1]. The MSU1.1 algorithm is a variant of MSU1 with three important modifications. First, it uses a better encoding for the one-hot constraints, namely a BDD representation of a counter converted to CNF with several optimizations. Second, MSU1.1 exploits the authors' observation that the one-hot constraints can actually be AtMost(1) constraints, because the identified cores are unsatisfiable if no clauses are removed. Third, an AtMost(1) constraint is placed on the clause-selector variables for each clause that has more than one.

The authors also describe MSU3, which avoids some of the size explosion of the additional variables and clauses created by MSU1 by using a single clause-selector variable per clause and a single AtMost constraint over all of them. In [11], the authors further introduce MSU4, essentially a modification of MSU3 that exploits relationships between unsatisfiable cores and bounds on Max-SAT solutions. Finally, Marques-Silva and Manquinho introduce MSU1.2 and MSU2

[1] We have adopted the algorithm naming from the most recent paper [9], which is slightly changed from earlier papers.

in [9]. MSU1.2 improves on MSU1.1 using a bitwise encoding, with a logarithmic number of auxiliary variables, for each cardinality constraint, and MSU2 takes that a step further, employing a bitwise one-hot encoding on the clause-selector variables themselves.

Table 1. Comparison of all MSU* algorithms

Algorithm	Cardinality Constraints	Cardinality Encoding
MSU1	Per-core One-Hot	Adder tree
MSU1.1	Per-core AtMost	BDD to CNF
MSU1.2	Per-core AtMost	Bitwise on variables
MSU2	Per-core One-Hot	Bitwise on clauses
MSU3	Single AtMost	BDD to CNF
MSU4-v1	Single AtMost	BDD to CNF
MSU4-v2	Single AtMost	Sorting networks

A parallel development of the concept of using unsatisfiable cores for Max-SAT was done in the domain of logic circuit debugging/diagnosis by Sülflow, et al. [15]. Without explicitly noting the connection to Max-SAT, they developed a new SAT-based debugging framework that exploits unsatisfiable core extraction. Though the terminology is different and the theories and algorithms are often described in terms of gates instead of constraints or clauses, SAT-based debugging is essentially the process of solving Max-SAT for circuit-derived CNF instances [12].

The primary difference between the work of Sülflow, et al. and the MSU* algorithms is that the debugging framework produces *all* Max-SAT results (equivalent to finding all minimum-cardinality MCSes) by an iterative solving procedure. Their use of cores is closest to MSU3, with a single cardinality constraint covering all identified cores; however, non-overlapping cores are given separate cardinality constraints, as this limits the size of the search space with little overhead. They do mention alternative approaches for producing cardinality constraints, including one which creates a separate constraint for every intersection of any subset of the cores, but they dismiss these as not outperforming their chosen approach in most of their experiments.

4 Using Cores to Find MCSes

Our algorithm is a synthesis of 1) the MCSes algorithm for finding all MCSes of an infeasible constraint system from [7] and 2) the application of unsatisfiable cores to the Max-SAT problem as first shown by Fu & Malik [5] and refined by Marques-Silva, et al. [9,10,11]. Because finding MCSes is a generalization of the Max-SAT problem (cf. Section 2), this combination is a natural one. In fact, the MCSes algorithm is very similar to the MSU3 algorithm described in [10].

MCSes-U(φ)

1. $k \leftarrow 1$ ◁ iteration counter
2. MCSes $\leftarrow \emptyset$ ◁ growing set of results
3. $\texttt{Core}_k \leftarrow$ **Core**(φ) ◁ any unsatisfiable core (preferably small) of φ
4. **while** (**InstrumentAll**(φ) + **Blocking**(MCSes)) **is satisfiable**
   ▽ clauses contained in $\texttt{Core}_k$ are instrumented with clause-selector variables
5. $\varphi_k \leftarrow$ **Instrument**($\varphi, \texttt{Core}_k$) + **AtMost**($\texttt{Core}_k, k$)
   ▽ **AllSAT** finds all models of φ_k corresponding to MCSes of size k
6. MCSes $\leftarrow$ MCSes + **AllSAT**(φ_k)
   ▽ the **Core** function projects instrumented clauses onto clauses of φ
7. $\texttt{Core}_{k+1} \leftarrow \texttt{Core}_k +$ **Core**($\varphi_k +$ **Blocking**(MCSes))
8. $k \leftarrow k + 1$
9. **return** MCSes

Fig. 1. The MCSes-U algorithm finds all MCSes of an unsatisfiable formula φ using unsatisfiable cores

Briefly, the overall approach of both MCSes and MSU3 is to instrument clauses in an unsatisfiable clause set with clause-selector variables, then to use cardinality constraints on those clause-selector variables to search for small subsets of clauses whose removal leaves the remaining set satisfiable. For Max-SAT, the goal is to find such a set of the smallest cardinality; finding MCSes requires finding all such sets that are minimal or irreducible. Therefore, it is reasonable to assume that an approach used to solve Max-SAT, especially one that has been paired with a basic algorithm so similar to that used for finding MCSes, could be applied to an algorithm for finding MCSes.

4.1 Algorithm

Figure 1 contains pseudocode for our algorithm, dubbed MCSes-U (the -U signifies its use of **u**nsatisfiable cores). Two persistent variables, k and MCSes, keep track of the current iteration and the set of results, respectively. In any particular iteration of the **do/while** loop, $\texttt{Core}_k$ contains the set of clauses that will be considered for removal, and thus potentially included in an MCS, in that iteration. The input formula is instrumented with clause-selector variables on those clauses contained within $\texttt{Core}_k$, and an AtMost constraint is added on those selector variables with the current bound k. The **AllSAT** function in MCSes-U behaves exactly like the incremental solving employed in MCSes: find a solution, record the MCS, block that MCS from future solutions with a blocking clause formed from its clause-selector variables, and continue until no solutions remain.

The core extraction in line 7 produces an unsatisfiable core of the combination of the instrumented formula φ_k with the blocking clauses produced from the set of MCSes found thus far (φ_k itself is satisfiable). This core is mapped back to clauses in the original clause set φ and added to $\texttt{Core}_k$ to make $\texttt{Core}_{k+1}$ for the following iteration. The process repeats as long as further MCSes remain, which can be determined by checking whether there is *any* way to make φ satisfiable by removing clauses *without* removing any MCS identified thus far.

For comparison purposes, consider that the previous algorithm MCSes is equivalent to MCSes-U under the condition that **Core** always returns the complete set of clauses in φ. In this situation, the entire formula will be instrumented with clause-selector variables in each iteration, and the AtMost bound will always apply to all of the clause-selector variables as well. The primary difference between MCSes and MCSes-U is that here we are using unsatisfiable cores to identify subsets of the clause set in which we know the MCSes must be found, or, conversely, we determine subsets that we know must *not* contain any MCSes. In the following subsection, we prove that this use of unsatisfiable cores is correct.

4.2 Completeness/Correctness Proof

Fu and Malik proved that their use of unsatisfiable cores in Max-SAT is correct in [5]; however, that proof does not carry over to our algorithm other than to prove that the first result returned will be a Max-SAT solution. We must further prove both 1) that every result returned by MCSes-U is an MCS (correctness) and 2) that all MCSes are found by the algorithm (completeness). These two points are interrelated:

Theorem 1. *Given an unsatisfiable clause set φ and a positive integer k:*
If all MCSes of φ of size less *than k are found, then every result of size k returned by MCSes-U(φ) is an MCS of φ.*

This is stated without a formal proof, but it follows from the correctness of the underlying algorithm for finding MCSes, described fully in [7], that we have adapted in this work. Briefly, the algorithm finds MCSes in increasing order of size; as every MCS of a size less than k is found, it is blocked from future solutions, and any correction set of size k that is found then must be minimal. With this theorem, we see that the algorithm's correctness hinges on its completeness. We shall prove that MCSes-U is complete in the following.

We wish to prove that the algorithm produces all MCSes of an instance. We will presuppose the completeness of the base algorithm as described in [7] and focus on the effect of our use of unsatisfiable cores. The base algorithm is equivalent to that presented in Figure 1 if we take $\texttt{Core}_k$ to be the complete formula φ in every iteration of the **while** loop (i.e., with no limitation on the clauses considered for finding MCSes). Therefore, we will prove here that the MCSes-U algorithm is complete in that it does not miss any MCSes due to restricting the search for MCSes to the clauses in $\texttt{Core}_k$.

First, we must define a new term, "k-correction," and present a useful lemma linking k-corrections to MCSes.

Definition 1. A k-correction of a set of clauses C is a set of k or fewer clauses whose removal makes C satisfiable.

Lemma 1. *Given an unsatisfiable subset C of a clause set φ and an integer k: If every $(k-1)$-correction of C contains some MCS of φ, then C contains all MCSes of φ with size k.*

(Proofs of this and the following lemmas are included in Appendix A.)

With this lemma, we can prove the completeness of our algorithm by induction. We wish to prove that the MCSes-U algorithm finds all MCSes of size k in the k^{th} iteration of its loop. We will first prove by induction that every $(k-1)$-correction of $\texttt{Core}_k$ contains an MCS of φ. Then, using Lemma 1, we can directly show that $\texttt{Core}_k$ contains all MCSes of size k, for all k. First, we will prove the base case of the inductive portion of our proof, for $k = 1$.

Lemma 2. *In MCSes-U, every 0-correction of $\texttt{Core}_1$ contains an MCS of φ.*

With Lemmas 1 and 2, we see that the algorithm is complete for $k = 1$. $\texttt{Core}_1$ contains all single-clause MCSes of φ, and the algorithm produces all MCSes of size 1. This can be seen from a different perspective by noting that an MCS of size 1 is a single clause, c, contained in every MUS of a formula, and thus $\texttt{Core}_k$, which is some unsatisfiable core of φ, must contain every MCS of size 1.

With our base case proven in Lemma 2, we now prove the inductive step.

Lemma 3. *Given some positive integer k:*
In MCSes-U, if every $(k-1)$-correction of $\texttt{Core}_k$ contains an MCS of φ, then every k-correction of $\texttt{Core}_{k+1}$ contains an MCS of φ.

With these lemmas, we can prove the completeness of MCSes-U in Theorem 2, which, with Theorem 1, proves that it is correct as well.

Theorem 2. *For any positive integer k: the MCSes-U algorithm finds all MCSes of size k in the k^{th} iteration of its loop.*

Proof. By Lemmas 2 and 3, we have that every $(k-1)$-correction of $\texttt{Core}_k$ contains an MCS of φ, for all k. With Lemma 1, then, $\texttt{Core}_k$ contains every MCS of φ of size k for all k. □

5 Experimental Results

Our primary experimental goal was to determine the value of using unsatisfiable cores to guide the search for MCSes in practice; specifically, we wished to compare the performance of MCSes and MCSes-U on industrial instances. In the course of running these experiments, we noticed an interesting situation in which using cores was in fact detrimental to the performance of Max-SAT algorithms but the MCSes-U algorithm still benefited, and we explore this case here as well.

Experimental Setup: All experiments were run in Linux (Fedora 9) on a 3.0GHz Intel Core 2 Duo E6850 with 3GB of physical RAM. The MCSes and MCSes-U algorithms were implemented in C++ using MiniSAT version 1.12b [4], which allows "native" AtMost constraints (instead of CNF encodings thereof). We added unsatisfiable core extraction to this version of MiniSAT using the resolution-graph method [17], storing the parents of each learned clause in memory. Binaries for MSU1.1 and MSU1.2 were supplied by João Marques-Silva.

Benchmark Families: We selected four sets of unsatisfiable industrial CNF benchmarks for these experiments:

- **Diagnosis:** These 108 instances, from the Max-SAT 2008 Evaluation [2], are generated in a process that diagnoses potential error locations in a physical circuit that is producing incorrect output(s) [12]. In this application, the MCSes of each instance directly identify the candidate error locations. (The set used in the Max-SAT Evaluation has 112 instances, and we removed 4 that are satisfiable.)
- **Reveal:** Reveal is a system for logic circuit verification that operates on Verilog code with a counterexample-guided abstraction refinement flow [1]. These 62 instances were generated in the abstraction refinement phase of Reveal when run on three different designs.
- **FVP-UNSAT.2.0:** 21 instances, used in previous SAT competitions, obtained from [16], and "generated in the formal verification of correct superscalar microprocessors."
- **DC:** This is a set of 84 instances from an automotive product configuration domain [13,14] that have previously been shown to have a wide range of characteristics with respect to each instance's MCSes and MUSes. They are of interest mainly because their diversity of results (from small sets found in less than a second to intractably large sets) exercises algorithms broadly.

The value of using cores is evident when we look at the results for finding multiple MCSes across all of these instances. Because the complete set of MCSes can be intractably large, we look at the *velocity* of finding MCSes: the number of MCSes found per second until all have been found or until a set timeout (600 seconds, here) has been reached. Many applications do not require the complete set of MCSes: the diagnosis task in [12] finds MCSes up to a certain cardinality, and the CAMUS algorithm used in Reveal can use a subset of the MCSes to find a subset of the MUSes of an instance [7]. Figure 2 compares the velocity of MCSes (w/o cores) to that of MCSes-U (w/ cores) on these instances. Points above the diagonal are instances where MCSes-U finds MCSes more quickly. MCSes-U outperforms MCSes in nearly all cases. With the Diagnosis instances in particular, we see several benchmarks for which MCSes finds no MCSes within the timeout, while MCSes-U outputs up to several hundred per second.

An interesting situation is displayed in Figure 3, which compares the runtime of the MCSes algorithm solving Max-SAT against three Max-SAT algorithms that use unsatisfiable cores: MCSes-U in the same Max-SAT mode, MSU1.1,

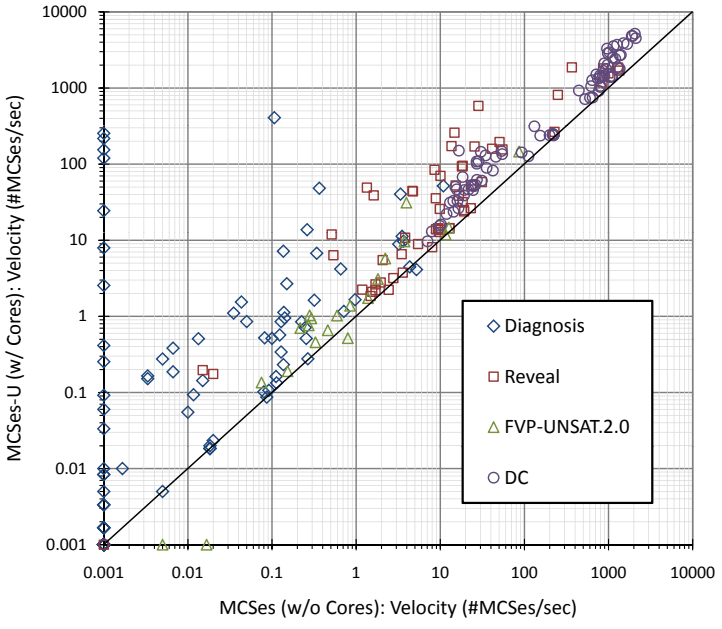


Fig. 2. Comparing the performance of MCSes and MCSes-U on industrial benchmarks. (600 second timeout, 0 velocity mapped to 0.001).

and MSU1.2 (these were the two MSU* algorithms with implementations available to us). For MCSes and MCSes-U to solve Max-SAT, they can both return the first MCS found and stop. Because the algorithms generate MCSes in increasing order of size, the first result is guaranteed to be an optimal Max-SAT solution; the complement of the MCS will be a maximum cardinality satisfiable subset of the constraints. Note that the MSU* binaries had a different implementation, using a different solver and CNF-encoded cardinality constraints, than MCSes[-U], so the results are not *directly* comparing the underlying algorithms.

These results are for the FVP-UNSAT.2.0 benchmarks. For these instances, we see that all of the algorithms that use cores take about two orders of magnitude longer than the vanilla MCSes algorithm. The number of Max-SAT solutions in each benchmark is large. For example, for each of the three 2pipe* instances in this set, approximately one quarter of the clauses are single-clause MCSes; removing any one of them makes the instance satisfiable. Therefore, solving Max-SAT for these instances is simple, as there are so many solutions, and they will be found in the first iteration of MCSes. Interestingly, the time taken to run a solver on each benchmark (in order to extract a core) far outweighs that taken to identify a single clause whose removal yields satisfiability.

We see that the time used to find a core can outweigh that needed to find a single MCS or Max-SAT solution in instances like the FVP-UNSAT.2.0 set with very many, single-clause MCSes. However, when finding MCSes, the overhead

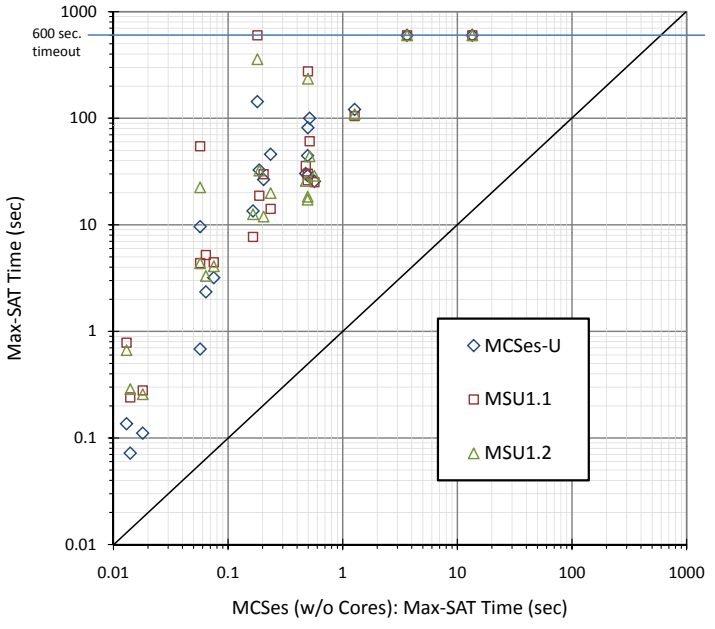


Fig. 3. Comparing the performance of MCSes solving Max-SAT against MCSes-U MSU1.1, and MSU1.2 on industrial benchmarks. (FVP-UNSAT.2.0 benchmarks.)

of finding cores is amortized over the large number of results and is outweighed by the increase in velocity gained from limiting the search space, even in those instances that appear to be worst-case scenarios for exploiting cores. Therefore, core extraction appears to be a safe addition to MCS algorithms with potentially large performance gains in industrial instances.

6 Conclusion

We have presented a generalization of the core-guided Max-SAT approach, applying it to the more general problem of identifying minimal correction sets (MCSes). By using unsatisfiable cores to guide the search for MCSes in a similar manner to their use in Max-SAT [5,9,10,11], we have realized significant performance gains in MCSes-U, an enhancement of the algorithm for finding MCSes in [7]. Experimental results show the value of this approach on a variety of industrial instances; it is particularly effective on instances generated by a circuit diagnosis application in which MCSes have a direct application.

Looking forward, we see that there are further ideas from the Max-SAT domain that can be applied to MCS algorithms. Notably, the use of one AtMost constraint per identified core, as in the MSU1.* algorithms, may be applicable to MCSes-U. For Max-SAT, the MSU1.* approach has shown better performance than the approach used in MSU3 and MSU4 of creating a single monolithic AtMost constraint over all extracted cores, and it may be beneficial for MCSes-U as

well. As with the proofs in this paper, determining and proving the correct application of the concept to the generalized problem of finding MCSes may require non-trivial work. We are also interested in investigating the combination and interplay of the core-guidance technique with autarky pruning, another method for reducing the search space of the MCS search [8].

Further, the results here motivate applying MCSes-U in circuit debugging / diagnosis, as MCSes was applied in [12]. While MCSes was used as an approximating preprocessor for an exact search in that work, the improved performance of MCSes-U may make it suitable for solving problems directly. A comparison to the algorithm in [15] could be instructive as well; though it is algorithmically very similar to MCSes-U, any substantial performance differences would indicate important implementation details that would aid in engineering future implementations. Further, [15] is restricted to only find minimum-cardinality solutions, and the more complete view of examining the set of *all* MCSes in such instances, which MCSes-U enables, could be beneficial.

Acknowledgments

We thank João Marques-Silva for providing binaries for his MSU* algorithms. This material is based upon work supported by the National Science Foundation under ITR Grant No. 0205288. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the National Science Foundation (NSF). This work was funded in part by the DARPA/MARCO Gigascale Systems Research Center.

References

1. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Refinement strategies for verification methods based on datapath abstraction. In: Proceedings of the 2006 conference on Asia South Pacific design automation (ASP-DAC 2006), pp. 19–24 (2006)
2. Argelich, J., Li, C.M., Manyà, F., Planes, J.: Max-SAT evaluation (2008), http://www.maxsat.udl.es/08/
3. Birnbaum, E., Lozinskii, E.L.: Consistent subsets of inconsistent systems: structure and behaviour. Journal of Experimental and Theoretical Artificial Intelligence 15, 25–46 (2003)
4. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
5. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 252–265. Springer, Heidelberg (2006)
6. Liffiton, M.H., Sakallah, K.A.: On finding all minimally unsatisfiable subformulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 173–186. Springer, Heidelberg (2005)
7. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning 40(1), 1–33 (2008)

8. Liffiton, M.H., Sakallah, K.A.: Searching for autarkies to trim unsatisfiable clause sets. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 182–195. Springer, Heidelberg (2008)
9. Marques-Silva, J., Manquinho, V.: Towards more effective unsatisfiability-based maximum satisfiability algorithms. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 225–230. Springer, Heidelberg (2008)
10. Marques-Silva, J., Planes, J.: On using unsatisfiability for solving maximum satisfiability. Computing Research Repository, abs/0712.1097 (December 2007)
11. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Proceedings of the Conference on Design, Automation, and Test in Europe (DATE 2008) (March 2008)
12. Safarpour, S., Liffiton, M., Mangassarian, H., Veneris, A., Sakallah, K.: Improved design debugging using maximum satisfiability. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD 2007), November 2007, pp. 13–19 (2007)
13. Sinz, C.: SAT benchmarks from automotive product configuration, http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/
14. Sinz, C., Kaiser, A., Küchlin, W.: Formal methods for the validation of automotive product configuration data. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17(1), 75–97 (2003)
15. Sülflow, A., Fey, G., Bloem, R., Drechsler, R.: Using unsatisfiable cores to debug multiple design errors. In: Proceedings of the 18th ACM Great Lakes symposium on VLSI, pp. 77–82 (2008)
16. Velev, M.: Miroslav Velev's SAT Benchmarks, http://www.miroslav-velev.com/sat_benchmarks.html
17. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In: The 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003) (2003)

A Proofs

Lemma 1. *Given an unsatisfiable subset C of a clause set φ and an integer k:*
If every $(k-1)$-correction of C contains some MCS of φ, then C contains all MCSes of φ with size k.

Proof. By contradiction: Assume that there exists some MCS M of φ with size k that is *not* contained entirely within C. We will denote the subset of M contained within C by $M' = M \cap C$. Thus, our assumption requires $|M'| \leq k-1$.

Because M is an MCS of φ and C is a subset of φ, M' must be a correction set of C. Formally, if $\varphi - M$ is satisfiable, then $C \cap (\varphi - M)$ must be as well. This can be transformed:

$$C \cap (\varphi - M) \;=\; (\varphi \cap C) - (M \cap C) \;=\; C - M'$$

And so M' is a correction set of C, because $C - M'$ is satisfiable.

Furthermore, M' is a $(k-1)$-correction of C, because $|M'| \leq k-1$. By the antecedent of this lemma, we know that M' must contain some MCS of φ. Because M is a proper superset of M', which contains an MCS, M can not be a *minimal* correction set of φ. This is a contradiction, and therefore we have proven that any MCS M of φ with size k must be contained entirely within C. □

Lemma 2. *In MCSes-U, every 0-correction of* $\texttt{Core}_1$ *contains an MCS of* φ.

Proof. Unsatisfiable clause sets have no 0-corrections, as removing 0 clauses can not make them satisfiable. $\texttt{Core}_1$ is an unsatisfiable clause set; therefore, $\texttt{Core}_1$ has no 0-corrections, and the lemma is trivially true. □

Lemma 3. *Given some positive integer* k*:*
In MCSes-U, if every $(k-1)$*-correction of* $\texttt{Core}_k$ *contains an MCS of* φ*, then every* k*-correction of* $\texttt{Core}_{k+1}$ *contains an MCS of* φ.

Proof. Proof by cases, depending on the k-corrections of $\texttt{Core}_k$:

Case 1: $\texttt{Core}_k$ has no k-corrections.
The algorithm includes $\texttt{Core}_k$ in $\texttt{Core}_{k+1}$. Therefore, in this case, $\texttt{Core}_{k+1}$ will have no k-corrections, as it is a superset of $\texttt{Core}_k$. Thus, trivially, every k-correction of $\texttt{Core}_{k+1}$ contains an MCS of φ.

Case 2: Every k-correction of $\texttt{Core}_k$ contains an MCS of φ.
Again, due to the fact that $\texttt{Core}_k \subseteq \texttt{Core}_{k+1}$, every k-correction of $\texttt{Core}_{k+1}$ is also a k-correction of $\texttt{Core}_k$, and thus every k-correction of $\texttt{Core}_{k+1}$ must contain some MCS of φ.

Case 3: At least one k-correction, δ, of $\texttt{Core}_k$ contains no MCSes of φ.
Because δ does not contain any MCSes of φ, the blocking clauses added to φ_k based on the MCSes of φ will all allow the relaxation of the clauses in δ. We will say that δ is thus an *unblocked* k-correction. At line 6 of MCSes-U, there exists a complete assignment for φ_k that relaxes all MUSes contained within $\texttt{Core}_k$ without violating the AtMost bound on relaxed clauses; such an assignment can relax the clauses in any unblocked k-correction.

However, φ_k is unsatisfiable at this point, after the addition of all blocking clauses for the MCSes found thus far (up to size k). Therefore, for any complete assignment that satisfies the blocking clauses and relaxes all MUSes contained in $\texttt{Core}_k$, there must be some MUS of φ that is not relaxed by that assignment. Any unsatisfiable core of φ_k will necessarily include one MUS of φ that is not relaxed for every such assignment. That is, any unblocked k-correction δ of $\texttt{Core}_k$ must be "counteracted" by including in $\texttt{Core}_{k+1}$ an MUS of φ untouched by δ.

Any k-correction of $\texttt{Core}_{k+1}$ must contain a k-correction of $\texttt{Core}_k$, because $\texttt{Core}_k \subseteq \texttt{Core}_{k+1}$. Any unblocked k-correction of $\texttt{Core}_k$ necessarily leaves at least one MUS in $\texttt{Core}_{k+1}$ untouched (by the construction of $\texttt{Core}_{k+1}$ described above). Thus, unblocked k-corrections of $\texttt{Core}_k$ cannot be k-corrections of $\texttt{Core}_{k+1}$. Therefore, the only k-corrections of $\texttt{Core}_{k+1}$ must be "unblocked," containing an MCS of φ.

These cases cover all possibilities, and, in every case, every k-correction of $\texttt{Core}_{k+1}$ contains an MCS of φ. □

Algorithms for Weighted Boolean Optimization

Vasco Manquinho[1], Joao Marques-Silva[2], and Jordi Planes[3]

[1] IST/UTL - INESC-ID
vasco.manquinho@inesc-id.pt
[2] University College Dublin
jpms@ucd.ie
[3] Universitat de Lleida
jplanes@diei.udl.cat

Abstract. The Pseudo-Boolean Optimization (PBO) and Maximum Satisfiability (MaxSAT) problems are natural optimization extensions of Boolean Satisfiability (SAT). In the recent past, different algorithms have been proposed for PBO and for MaxSAT, despite the existence of straightforward mappings from PBO to MaxSAT, and vice-versa. This papers proposes Weighted Boolean Optimization (WBO), a new unified framework that aggregates and extends PBO and MaxSAT. In addition, the paper proposes a new unsatisfiability-based algorithm for WBO, based on recent unsatisfiability-based algorithms for MaxSAT. Besides standard MaxSAT, the new algorithm can also be used to solve weighted MaxSAT and PBO, handling pseudo-Boolean constraints either natively or by translation to clausal form. Experimental results illustrate that unsatisfiability-based algorithms for MaxSAT can be orders of magnitude more efficient than existing dedicated algorithms. Finally, the paper illustrates how other algorithms for either PBO or MaxSAT can be extended to WBO.

1 Introduction

In the area of Boolean-based decision and optimization procedures, natural extensions of Boolean Satisfiability (SAT) include Maximum Satisfiability (MaxSAT) [10] and Pseudo-Boolean Optimization (PBO) [6]. Algorithms for MaxSAT and PBO have been the subject of significant improvements over the last few years. This in turn, motivated the use of both PBO and, more recently, of MaxSAT in a number of practical applications. Interestingly, albeit there are simple translations from any MaxSAT variant to PBO and vice-versa (by encoding to CNF) [1, 18], algorithms for MaxSAT and PBO have evolved separately, and often use fairly different algorithmic organizations. Nevertheless, there exists work that acknowledges this relationship and algorithms that can solve instances of MaxSAT and of PBO have already been proposed [1, 18].

Recent work has provided more alternatives for solving either MaxSAT or PBO, by using SAT solvers and the identification of unsatisfiable sub-formulas [16, 27]. However, the proposed algorithms were restricted to the plain and partial variants of MaxSAT and to a restricted form of Binate Covering for PBO. This paper extends this recent work in a number of directions. First, the paper proposes a simple algorithm for (Partial) Weighted MaxSAT, using unsatisfiable sub-formula identification. Second, the paper generalizes MaxSAT and PBO by introducing Weighted Boolean Optimization (WBO),

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 495–508, 2009.

a new modeling framework for solving linear optimization problems over Boolean domains. Third, the paper shows how to extend the unsatisfiability-based algorithm for MaxSAT for solving WBO problems. Finally, the paper suggests how other algorithms can be used for solving WBO. Besides the proposed contributions, the paper also provides empirical evidence that unsatisfiability-based MaxSAT and WBO solvers can outperform state-of-the-art solvers on problem instances from practical problems.

The paper is organized as follows. Section 2 provides a brief overview of the topics addressed in the paper, namely MaxSAT, PBO, translations from MaxSAT to PBO and vice-versa, and unsatisfiability-based algorithms for MaxSAT. Section 3 details an algorithm for (Partial) Weighted MaxSAT based on unsatisfiable sub-formula identification. Next, Section 4 introduces Weighted Boolean Optimization (WBO), and shows how to extend the algorithm of Section 3 to WBO. Section 5 analyzes the experimental results, obtained on representative classes of problem instances. Section 6 overviews related work, and Section 7 concludes the paper.

2 Preliminaries

This section briefly introduces the Maximum Satisfiability (MaxSAT) problem and its variants, as well as the Pseudo-Boolean Optimization (PBO) problem. The main approaches used by state-of-the-art solvers are summarized. Moreover, translation procedures from MaxSAT to PBO and vice-versa are overviewed. Finally, unsatisfiability-based MaxSAT algorithms are surveyed, all of which the paper uses in later sections.

2.1 Maximum Satisfiability

Given a CNF formula φ, the Maximum Satisfiability (MaxSAT) problem can be defined as finding an assignment that maximizes the number of satisfied clauses (which implies that the assignment minimizes the number of unsatisfied clauses). Besides the classical MaxSAT problem, there are also three well-known variants of MaxSAT: weighted MaxSAT, partial MaxSAT and weighted partial MaxSAT. All these formulations have been used in a wide range of practical applications, namely scheduling, FPGA routing [34], design automation [31], among others.

A partial CNF formula is described as the conjunction of two CNF formulas φ_h and φ_s, where φ_h represents the *hard* clauses and φ_s represents the *soft* clauses. The *partial* MaxSAT problem consists of finding an assignment to the problem variables such that all hard clauses (φ_h) are satisfied, and the number of satisfied soft clauses (φ_s) is maximixed.

A weighted CNF formula is a set of weighted clauses. A weighted clause is a pair (ω, c), where ω is a classical clause and c is a positive natural number ($\mathbb{N}^*$) corresponding to the cost of unsatisfying ω. Given a weighted CNF formula, the *weighted* MaxSAT problem consists of finding an assignment to the problem variables such that the total weight of the satified clauses is maximized (which implies that the total weight of the unsatisfied clauses is minimized).

A weighted partial CNF formula is the conjunction of a weighted CNF formula (soft clauses) and a classical CNF formula (hard clauses). The *weighted partial* MaxSAT

problem consists of finding an assignment to the variables such that all hard clauses are satisfied and the total weight of satisfied soft clauses is maximized. Observe that, for both partial MaxSAT and weighted partial MaxSAT, hard clauses can also be represented as weighted clauses: one can consider that the weight is greater than the sum of the weights of the soft clauses.

Starting with the seminal work of Borchers and Furman [10], there has been an increasing interest in developing efficient MaxSAT solvers. Following such work, two branch and bound based solvers have been developed: (i) MaxSatz [20], the first solver to implement a unit propagation based lower bound and a failed literal based lower bound, both closely linked with a set of inference rules. Such a solver has been extended into several solvers: IncMaxSatz [22] and MaxSatz$_{icss}$ [13], with efficient incremental lower bound computation; and WMaxSatz [3], which deals with weighted clauses; (ii) MiniMaxSAT [18], a solver created on top of MiniSAT with MaxSAT resolution [9] applied over an unsatisfiable sub-formula detected by the unit propagation based lower bound. A different approach has been the conversion of MaxSAT into a different formalism. The most notable works using this approach have been: Toolbar [19], a weighted CSP solver which converts MaxSAT instances into a weighted constraint network; SAT4J MAXSAT [7], a solver which iteratively converts a MaxSAT instance into a PBO instance; Clone [29] and sr(w) [30], solvers which convert a MaxSAT instance into a deterministic decomposable negation normal form (d-DNNF) instance; and MSUnCore [27], a solver which solves MaxSAT using the unsatisfiable cores detected by iteratively encoding the problem instance into SAT. In the last Max-SAT Evaluation [4], this latter approach has been shown to be effective for industrial problems.

2.2 Pseudo-Boolean Optimization

The Pseudo-Boolean Optimization (PBO) problem is another extension of SAT where constraints can be any linear inequality with integer coefficients (also known as pseudo-Boolean constraints) defined over the set of problem variables. The objective in PBO is to find an assignment to problem variables such that all problem constraints are satisfied and the value of a linear objective function is optimized. Any pseudo-Boolean formulation can be easily translated into a normal form [6] such that all integer coefficients are non-negative.

$$\begin{aligned} &\text{minimize} \quad \sum_{j \in N} c_j \cdot x_j \\ &\text{subject to} \quad \sum_{j \in N} a_{ij} l_j \geq b_i, \\ &\qquad l_j \in \{x_j, \bar{x}_j\}, x_j \in \{0,1\}, a_{ij}, b_i, c_j \in \mathbf{N}_0^+ \end{aligned} \tag{1}$$

Almost all algorithms to solve PBO rely on the generalization of the most effective techniques already used in SAT solvers, namely Boolean Constraint Propagation, conflict-based learning and conflict-directed backtracking [23,11]. Nevertheless, there are several approaches to solve PBO formulations. The most common using SAT solvers is to make a linear search on the value of the objective function. The idea is to generalize SAT algorithms to deal natively with pseudo-Boolean constraints [6] and whenever

a solution for the problem constraints is found, a new constraint is added such that only solutions with a lower value for the objective function can be accepted. The algorithm finishes when the solver cannot improve on the last solution found, therefore proving its optimality.

Another common approach is branch and bound, where lower bounding procedures to estimate the value of the objective function are used. Several lower bounding procedures have been proposed, namely Maximum Independent Set of constraints [12], Linear Programming Relaxation [21, 24], among others [24]. There are also algorithms that encode pseudo-Boolean constraints into propositional clauses [33, 5, 15] and solve the problem by subsequently using a SAT solver. This approach has been proved to be very effective for several problem sets, in particular when the clause encoding is not much larger than the original pseudo-Boolean formulation.

2.3 Translations between MaxSAT and PBO

Although MaxSAT and PBO are different formalisms, it is possible to encode any MaxSAT instance into a PBO instance and vice-versa [2, 1, 17]. This section focus solely on weighted partial MaxSAT, since the encodings of the other variants easily follow.

The encoding of hard clauses from weighted partial MaxSAT to PBO is straightforward, since propositional clauses are a particular case of pseudo-Boolean constraints. However, for each soft clause $\omega_i = (l_1 \vee l_2 \vee \ldots \vee l_k)$ with weight c_i, the encoding to PBO involves the use of an additional selection variable s_i, such that the corresponding constraint in PBO to ω_i would be $s_i + \sum_{j=1}^{k} l_j \geq 1$. This ensures that variable s_i is assigned to true whenever ω_i is not satisfied. The objective function of the corresponding PBO instance is to minimize the weighted sum of the selection variables. For each selection variable s_i in the objective function, its coefficient is the weight c_i of the corresponding soft clause ω_i.

Example 1. Consider the following weighted partial MaxSAT instance.

$$\begin{array}{l} \varphi_h = \{\ (x_1 \vee x_2 \vee \bar{x}_3), (\bar{x}_2 \vee x_3), (\bar{x}_1 \vee x_3)\} \\ \varphi_s = \{\ (\bar{x}_3, 6), (x_1 \vee x_2, 3), (x_1 \vee x_3, 2)\} \end{array} \tag{2}$$

According to the described encoding, the corresponding PBO instance would be:

$$\begin{array}{rr} \text{minimize} & 6s_1 + 3s_2 + 2s_3 \\ \text{subject to} & x_1 + x_2 + \bar{x}_3 \geq 1 \\ & \bar{x}_2 + x_3 \geq 1 \\ & \bar{x}_1 + x_3 \geq 1 \\ & s_1 + \bar{x}_3 \geq 1 \\ & s_2 + x_1 + x_2 \geq 1 \\ & s_3 + x_1 + x_3 \geq 1 \end{array} \tag{3}$$

The encoding of PBO constraints into MaxSAT can be done using any of the proposed encodings from pseudo-Boolean constraints to clauses [33, 5, 15]. Hence, for each pseudo-Boolean constraint there will be a set of hard clauses encoding it in the respective MaxSAT instance. The number of clauses and additional variables, depends on the

translation process used. The encoding is trivial when the original constraint in the PBO instance is already a clause.

The objective function of PBO instances can be encoded into MaxSAT with the use of weighted soft clauses. The idea is that for each variable x_j with coefficient c_j in the objective function, a corresponding soft clause $(\bar{x}_j)$ with weight c_j is added to the MaxSAT instance. Therefore, the solution of the MaxSAT formulation minimizes the weighted sum of problem variables, as required in the PBO instance.

Example 2. For illustration purposes, consider the following PBO instance:

$$\begin{array}{ll} \text{minimize} & 4x_1 + 2x_2 + x_3 \\ \text{subject to} & 2x_1 + 3x_2 + 5x_3 \geq 5 \\ & \bar{x}_1 + \bar{x}_2 \geq 1 \\ & x_1 + x_2 + x_3 \geq 2 \end{array} \tag{4}$$

Note that the first and third constraint must be encoded into CNF, but the second constraint is already a clause and so it can be represented directly as a hard clause. The corresponding MaxSAT instance would be:

$$\begin{array}{l} \varphi_h = \{ \text{CNF}(2x_1 + 3x_2 + 5x_3 \geq 5), (\bar{x}_1 \vee \bar{x}_2), \text{CNF}(x_1 + x_2 + x_3 \geq 2)\} \\ \varphi_s = \{ (\bar{x}_1, 4), (\bar{x}_2, 2), (\bar{x}_3, 1)\} \end{array} \tag{5}$$

2.4 Unsatisfiability-Based MaxSAT

Recent work proposed the use of SAT solvers to solve (partial) MaxSAT, by iteratively identifying and relaxing unsatisfiable sub-formulas [16,27,26,25]. In this paper we refer to these algorithms generically as MSU (Maximum Satisfiability with Unsatisfiability) algorithms.

The original algorithm of Fu&Malik [16] (referred to as MSU1.0) iteratively identifies unsatisfiable sub-formulas. For each computed unsatisfiable sub-formula, all original (soft) clauses are relaxed with fresh relaxation variables. Moreover, a new Equals1 (or AtMost1) constraint relates the relaxation variables of each iteration, i.e. exactly 1 of these relaxation variables can be assigned value 1. The MSU1.0 algorithm can use more than one relaxation variable for each clause. In the original algorithm [16], a quadratic pairwise encoding of the Equals1 constraint was used. Finally, observe that the Equals1 constraint in line 13 of Algorithm 1 can be replaced by an AtMost1 constraint, without affecting the correctness of the algorithm.

More recently, several new MSU algorithms were proposed [26,27]. The differences of the MSU algorithms include the number of cardinality constraints used, the encoding of cardinality constraints (of which the AtMost1 and Equals1 constraints are a special case), the number of relaxation variables considered for each clause, and how the MSU algorithm proceeds. Extensive experimentation (from [25] but also from the MaxSAT Evaluation [4]) suggests that an optimized variation of Fu&Malik's algorithm[25] is currently the best performing MSU algorithm.

3 Unsatisfiability-Based Weighted MaxSAT

This section describes extensions of MSU1.0, described in Algorithm 1, for solving (Partial) Weighted MaxSAT problems. One simple solution is to create c_j replicas of

Algorithm 1. The (Partial) MaxSAT algorithm of Fu&Malik [16]

```
MSU1(φ)
1  φ_W ← φ                                          ▷ Working formula, initially set to φ
2  while true
3      do (st, φ_C) ← SAT(φ_W)
4         ▷ φ_C is an unsatisfiable sub-formula if φ_W is unsat
5         if st = UNSAT
6           then V_R ← ∅
7                for each ω ∈ φ_C
8                    do if not hard(ω)
9                         then r is a new relaxation variable
10                             ω_R ← ω ∪ {r}
11                             φ_W ← φ_W − {ω} ∪ {ω_R}
12                             V_R ← V_R ∪ {r}
13               φ_R ← CNF(∑_{r∈V_R} r = 1)          ▷ Equals1 constraint
14               Set all clauses in φ_R as hard clauses
15               φ_W ← φ_W ∪ φ_R                     ▷ Clauses in φ_R are declared hard
16          else ▷ Solution to MaxSAT problem
17               ν ← | relaxation variables w/ value 1 |
18               return |φ| − ν
```

clause ω_j, where c_j is the weight of clause ω_j. The resulting extended CNF formula can then be solved by MSU1.0. The proof of Fu&Malik's paper would also apply in this case, and so correctness follows. The operation of this solution for (Partial) Weighted MaxSAT justifies a few observations. Consider an unsatisfiable sub-formula φ_C where the smallest weight is min_c. Each clause would be replaced by a number of replicas. Hence, this unsatisfiable sub-formula would be identified min_c times. Clearly, this solution is unlikely to scale for clauses with very large weights. Hence, a more effective solution is needed, which is detailed below.

An alternative solution is to split a clause only when the clause is included in an unsatisfiable sub-formula. The way the clause is split depends on its weight. An algorithm implementing this solution is shown in Algorithm 2. For each unsatisfiable sub-formula, the smallest weight min_c of the clauses in the sub-formula is computed. This smallest weight is then used to update a lower bound on minimum cost of unsatisfiable clauses. Clauses in the unsatisfiable sub-formula are relaxed. However, if the weight of a clause is larger than min_c, then the clause is split: a new relaxed clause with weight min_c is created, and the weight of the original clause is decreased by min_c.

Example 3. Consider the partial MaxSAT instance in (2). Assume that the unsatisfiable sub-formula detected in line 4 of Algorithm 2 is:

$$\varphi_C = \{\, (\bar{x}_2 \vee x_3), (\bar{x}_1 \vee x_3), \quad (\bar{x}_3, 6), (x_1 \vee x_2, 3) \,\}. \tag{6}$$

Algorithm 2. Unsatisfiability-based (Partial) Weighted MaxSAT algorithm : WMSU1

```
WMSU1(φ)
1  φ_W ← φ                                          ▷ Working formula, initially set to φ
2  cost_lb ← 0
3  while true
4      do (st, φ_C) ← SAT(φ_W)
5         ▷ φ_C is an unsatisfiable sub-formula if φ_W is unsat
6         if st = UNSAT
7           then min_c ← min_{ω∈φ_C ∧ hard(ω)} cost(ω)
8                cost_lb ← cost_lb + min_c
9                V_R ← ∅
10               for each ω ∈ φ_C
11                   do if not hard(ω)
12                        then r is a new relaxation variable
13                             V_R ← V_R ∪ {r}
14                             ω_R ← ω ∪ {r}
15                             cost(ω_R) ← min_c
16                             if cost(ω) > min_c
17                               then φ_W ← φ_W ∪ {ω_R}
18                                    cost(ω) ← cost(ω) − min_c
19                               else φ_W ← φ_W − {ω} ∪ {ω_R}
20               φ_R ← CNF(∑_{r∈V_R} r = 1)          ▷ Equals1 constraint
21               Set all clauses in φ_R as hard clauses
22               φ_W ← φ_W ∪ φ_R                     ▷ Clauses in φ_R are declared hard
23          else ▷ Solution to Weighted MaxSAT problem
24               return cost_lb
```

Then, the smallest weight min_c is 3, and the new formula becomes $\varphi_W = \varphi_h \cup \varphi_s$, where

$$\begin{array}{l} \varphi_h = \{\ (x_1 \vee x_2 \vee \bar{x}_3), (\bar{x}_2 \vee x_3), (\bar{x}_1 \vee x_3), \text{CNF}(s_1 + s_2 = 1)\ \} \\ \varphi_s = \{\ (\bar{x}_3, 3), (x_1 \vee x_3, 2), (s_1 \vee \bar{x}_3, 3), (s_2 \vee x_1 \vee x_2, 3)\ \}. \end{array} \tag{7}$$

Observe that the new algorithm can be viewed as a direct optimization of the naive algorithm outlined earlier. The main difference is that each iteration of the algorithm collapses min_c iterations of the naive algorithm. For clauses with large weights the difference can be significant.

Theorem 1 (Correctness of WMSU1). *The value returned by Algorithm 2 is the minimum cost of non-satisfied clauses in φ.*

Proof. The previous discussion and the proof in [16]. □

4 Weighted Boolean Optimization

This section introduces Weighted Boolean Optimization (WBO), a new framework for modeling with hard and soft pseudo-Boolean constraints, that extends both MaxSAT

and its variants and PBO. Furthermore, a new algorithm based on identifying unsatisfiable sub-formulas is also proposed for solving WBO.

A Weighted Boolean Optimization (WBO) formula φ is composed of two sets of pseudo-Boolean constraints, φ_s and φ_h, where φ_s contains the soft constraints and φ_h contains the hard constraints. For each soft constraint $\omega_i \in \varphi_s$ there is an associated integer weight $c_i > 0$. The WBO problem consists of finding an assignment to the problem variables such that all hard constraints are satisfied and the total weight of the unsatisfied soft constraints is minimized (i.e. the total weight of satisfied soft constraints is maximized).

It should be noted that WBO represents a generalization of weighted partial MaxSAT by introducing the use of pseudo-Boolean constraints instead of just using propositional clauses. Hence, more compact formulations can be obtained with WBO than with MaxSAT. Moreover, PBO formulations can also be linearly encoded into WBO. Constraints in PBO can be directly encoded as hard constraints in WBO and the objective function can also be encoded as described in section 2.3. Therefore, WBO is a generalization of MaxSAT and its variants, as well as of PBO, allowing a unified modeling framework to integrate both of these Boolean optimization problems.

4.1 Unsatisfiability-Based WBO

This section describes how Algorithm 2 (introduced in Section 3) for weighted partial MaxSAT can be modified for solving WBO formulas. First of all, in a WBO formula, constraints are not restricted to be propositional clauses. Both soft and hard constraints can be pseudo-Boolean constraints. Hence, φ is a pseudo-Boolean formula, instead of a CNF formula. Moreover, the use of a SAT solver in line 6 is replaced with a pseudo-Boolean solver extended with the ability to generate an unsatisfiable sub-formula from the original pseudo-Boolean formula.

Next, if the formula is unsatisfiable, the weight associated with the unsatisfiable sub-formula is computed in the same way (lines 9-13) and the soft constraints in the core must also be relaxed using new relaxation variables (lines 15-24). Consider that $\omega = \sum a_j l_j \geq b$ denotes the pseudo-Boolean constraint to be relaxed using variable r. The resulting relaxed constraint in line 19 will be $\omega_R = b \cdot r + \sum a_j l_j \geq b$.

Finally, the constraint on the new relaxation variables in line 25 does not need to be encoded into CNF. The pseudo-Boolean constraint $\sum_{r \in V_R} r = 1$ can be directly added to φ_W, resulting in a more compact formulation, in particular if the number of soft constraints in the core is large.

In some cases, for an unsatisfiable sub-formula with k soft constraints, it is possible to use less than k additional variables. Consider the following soft constraints $\omega_1 = \sum_{l_j \in L_1} a_{1j} l_j \geq b_1$ and $\omega_2 = \sum_{l_j \in L_2} a_{2j} l_j \geq b_2$ in a given unsatisfiable sub-formula, where L_1 and L_2 denote respectively the set of literals in constraints ω_1 and ω_2. Additionally, let $x_k \in L_1$, $\bar{x}_k \in L_2$, $a_{1k} \geq b_1$ and $a_{2k} \geq b_2$, i.e. assigning x_k to true satisfies ω_1 and assigning x_k to false satisfies ω_2.[1] In this case, these constraints can share the same relaxing variable. This is due to the fact that it is impossible for

[1] This is a generalization to pseudo-Boolean constraints. Note that if the WBO instance corresponds to a MaxSAT instance, this is very common to occur, since ω_1 and ω_2 are clauses.

both ω_1 and ω_2 to be unsatisfied by the same assignment, since either x_k satisfies ω_1 or $\bar{x}_k$ satisfies ω_2. Therefore, by using the same relaxing variable on both constraints, it is maintained the restriction that at most one soft constraint in the core can be relaxed.

Example 4. Suppose that the following set of soft constraints defines an unsatisfiable sub-formula in a WBO instance:

$$\begin{aligned} \omega_1 &= 2x_1 + 3x_2 + 5x_3 \geq 5 \\ \omega_2 &= \bar{x}_1 + \bar{x}_2 \geq 1 \\ \omega_3 &= x_2 + \bar{x}_3 \geq 1 \\ \omega_4 &= x_1 + \bar{x}_3 \geq 1 \end{aligned} \tag{8}$$

In this case, constraints ω_1 and ω_3 can share the same relaxation variable, since the assignment of a value to x_3 implies that either ω_1 or ω_3 is satisfied. The same occurs with ω_2 and ω_4, given that the assignment to x_1 either satisfies ω_2 or ω_4. Therefore, after the relaxation, the resulting formula can include just two relaxation variables, instead of four. The resulting formula would be:

$$\begin{aligned} 5s_1 + 2x_1 + 3x_2 + 5x_3 &\geq 5 \\ s_2 + \bar{x}_1 + \bar{x}_2 &\geq 1 \\ s_1 + x_2 + \bar{x}_3 &\geq 1 \\ s_2 + x_1 + \bar{x}_3 &\geq 1 \\ s_1 + s_2 &\leq 1 \end{aligned} \tag{9}$$

The application of this reduction rule of relaxing variables raises the problem of finding the smallest number of relaxation variables to be used. This problem can be mapped into finding a matching of maximum cardinality in an undirected graph. In such a graph, there is a vertex for each constraint in the unsatisfiable sub-formula, while edges connect vertexes corresponding to constraints that can share a relaxation variable. The problem of finding a matching of maximum cardinality in an undirected graph can be solved in polynomial time [14]. Nevertheless, our prototype implementation of WBO solver uses a greedy algorithmic approach.

4.2 Other Algorithms for WBO

An alternative solution for solving WBO is to extend existing PBO algorithms. For example, soft pseudo-Boolean constraints can be represented in a PBO instance as relaxable constraints, and the overall cost function becomes the weighted sum of the relaxation variables of all soft pseudo-Boolean constraints of the original WBO formulation. This solution resembles the existing approach for solving MaxSAT with PBO [2,1], and has the same potential drawbacks.

One additional alternative solution is to generalize branch and bound weighted partial MaxSAT solvers to deal with soft and hard pseudo-Boolean constraints. However, note that these approaches focus on a search process that uses successive refinements on the upper bound of the WBO solution, while the algorithm proposed in section 4.1 works by refining lower bounds on the optimum solution value.

5 Results

With the objective of evaluating the new (partial) weighted MaxSAT algorithm and the new WBO solver, a set of industrially-motivated problem instances was selected. The characteristics of the classes of instances considered are shown in Table 1. For each class of instances, the table provides the class name, the number of instances (#I), the type of MaxSAT variant, and the source for the class of instances.

Table 1. Classes of problem instances

Class	#I	MaxSAT Variant	Source
IND	110	Partial Weighted	MaxSAT Evaluation 2009
FIR	59	Partial	Pseudo-Boolean Evaluation 2007
SYN	74	Partial	Pseudo-Boolean Evaluation 2005

Moreover, a wide range of MaxSAT and PBO solvers were considered, all among the best performing in either the MaxSAT or the Pseudo-Boolean evaluations. The weighted MaxSAT solvers considered were WMaxSatz [3], MiniMaxSat [18], IncWMaxSatz [22], Clone [29], and SAT4J (MaxSAT) [7]. Solver sr(w) [30] was also considered, but the results are not competitive. In addition, a new version of MSUnCore [26,27,25], integrating the weighted MaxSAT algorithm proposed in Section 3, was also evaluated. The PBO solvers considered were BSOLO [24], PBS [1], Pueblo [32], Minisat+[15], and SAT4J (PB) [7]. Finally, results for the new WBO solver, implementing the WBO organization described in Section 4 is also shown.

All experiments were run on a cluster of Linux AMD Opteron 2GHz servers with 1GB of RAM. The CPU time limit was set to 1800 seconds, and the RAM limit was set to 1 GB.

All algorithms were run on all problem instances considered. The original representations were used, in order to avoid introducing any bias towards any of the problem representations. Tables 2 and 3 summarize the number of instances aborted by each solver for each class of instances. As can be concluded, for practical problem instances, only a small number of MaxSAT solvers is effective. The results are somewhat different for the PBO solvers, where several can be competitive for different classes of instances. It should be noted that the IND benchmarks can be considered challenging for pseudo-Boolean solvers due to the large clause weights used.

For class IND and for the MaxSAT solvers, the results are somewhat surprising. Some of the solvers perform extremely well, whereas the others cannot solve most of the problem instances. IncWMaxSatz, MSUnCore and WBO are capable of solving *all* problem instances, but other MaxSAT solvers abort the vast majority of the problem instances. One additional observation is the very good performance of IncWMaxSatz when compared to WMaxSatz. This clearly indicates that the lower bound computation used in IncWMaxSatz can be very effective, even for industrial problem instances. For the PBO solvers, given the set of benchmark instances considered, SAT4J (PB) and BSOLO come out as the best performing. Clearly, this conclusion is based on the class

Table 2. Solved Instances for MaxSAT Solvers

Class	WMaxSatz	MiniMaxSat	IncWMaxSatz	Clone	SAT4J (MS)	MSUncore
IND	11	0	110	0	10	110
FIR	7	14	33	5	10	45
SYN	22	29	19	13	21	34
Total (Out of 243)	40	43	162	18	41	189

Table 3. Solved Instances for PBO & WBO Solvers

Class	BSOLO	PBS	Pueblo	Minisat+	SAT4J (PB)	WBO
IND	17	0	0	0	60	110
FIR	20	11	14	22	7	39
SYN	51	19	30	30	22	33
Total (Out of 243)	88	30	44	52	89	182

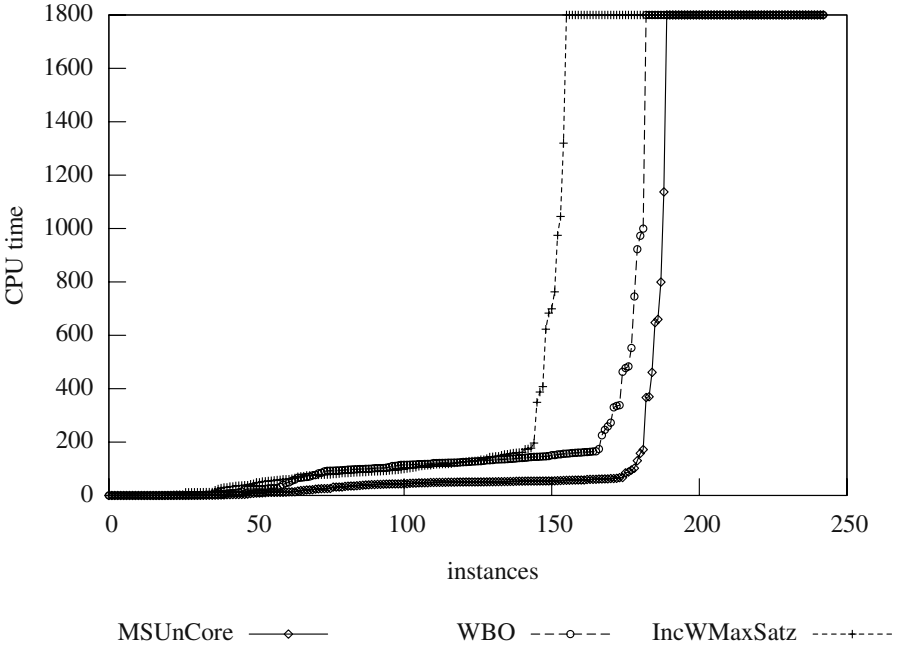


Fig. 1. Run times for IncWMaxSatz, MSUnCore, and WBO for all instances

of instances considered, which nevertheless derive from practical applications. Moreover, SAT4J (PB) performs significantly better than SAT4J (MaxSAT). This may be the result of a less effective encoding internally to SAT4J.

Motivated by the overall results, the best MaxSAT, PBO and the WBO solver were analyzed in more detail. Given the experimental results, IncWMaxSatz, MSUnCore,

and WBO were selected. Figure 1 shows the results for the selected solvers by increasing run times.

As can be concluded, the plot confirms the trends in the tables of results. MSUnCore is the best performing, followed by WBO and IncWMaxSatz. For smaller run times (instances from class IND), IncWMaxSatz can be more efficient than WBO. Moreover, these results indicate that, for the classes of instances considered, encoding cardinality constraints into CNF (as done in MSUnCore) may be a better solution than natively handling cardinality and pseudo-Boolean constraints (as done in WBO). It should be noted that all the instances considered can be encoded with cardinality constraints, for which existing polynomial encodings guarantee arc-consistency. This is not true for problem instances that use other pseudo-Boolean constraints, and for which encodings that ensure arc-consistency are exponential in the worst-case [15]. Finally, another source of difference in the experimental results is that whereas MSUnCore is built on top of PicoSAT [8], WBO is built on top of Minisat2. The different underlying SAT solvers may also contribute to explain some of the differences observed.

6 Related Work

A brief account of MaxSAT and PBO solvers is provided in Section 2. The use of unsatisfiability for solving MaxSAT was first proposed in 2006 [16]. This work was later extended [26,27,25], to accommodate several alternative algorithms and a number of optimizations to the first algorithm. To the best of our knowledge, MSUnCore is the first algorithm for solving (Partial) Weighted MaxSAT with unsatisfiable sub-formula identification. Also, to the best of our knowledge, WBO represents a new modeling framework, and the associate algorithm is new.

The use of optimization variants of decision procedures has also been proposed in the area of SMT [28], and a few SMT solvers now offer the ability for solving optimization problems. The approaches used for solving optimization problems in SMT are based on the use of relaxation variables, similarly to the PBO approach for solving MaxSAT [1].

7 Conclusions and Future Work

This paper proposes a new algorithm for (Partial) Weighted MaxSAT, based on unsatisfiable sub-formula identification. In addition, the paper introduces Weighted Boolean Optimization (WBO), that aggregates and generalizes PBO and MaxSAT. The paper then shows how unsatisfiability-based algorithms for (Partial) Weighted MaxSAT can be extended to WBO. Finally, the paper illustrates how to extend other algorithms for PBO and MaxSAT to solve WBO.

Experimental results, obtained on a representative set of benchmark instances with industrial motivations, shows that the new algorithm for weighted MaxSAT can outperform other existing algorithms by orders of magnitude. The experimental results also provide a preliminary (albeit possibly biased) study on the performance differences between handling pseudo-Boolean constraints natively and encoding to CNF. Finally, the paper shows that a general algorithm for WBO can be as efficient as other dedicated algorithms.

The integration of MaxSAT and PBO into a unique optimization extension of SAT increases the range of problems that can be solved. It also allows developing other general purpose algorithms, integrating the best techniques from both domains. Future research work will address adapting other algorithms for WBO. One concrete example is the use of PBO solvers. The other is extending the existing family of MSU algorithms for WBO.

Acknowledgement. This work is partially supported by EU grant ICT/217069 and FCT grant PTDC/EIA/76572/2006.

References

1. Aloul, F., Ramani, A., Markov, I., Sakallah, K.A.: Generic ILP versus specialized 0-1 ILP: An update. In: International Conference on Computer-Aided Design, pp. 450–457 (2002)
2. Amgoud, L., Cayrol, C., Berre, D.L.: Comparing arguments using preference ordering for argument-based reasoning. In: International Conference on Tools with Artificial Intelligence, pp. 400–403 (1996)
3. Argelich, J., Li, C.M., Manà, F.: An improved exact solver for partial max-sat. In: International Conference on Nonconvex Programming: Local and Global Approaches, pp. 230–231 (2007)
4. Argelich, J., Li, C.M., Manyà, F., Planes, J.: Third Max-SAT evaluation (2008), http://www.maxsat.udl.cat/08/
5. Bailleux, O., Boufkhad, Y., Roussel, O.: A translation of pseudo Boolean constraints to SAT. Journal on Satisfiability, Boolean Modeling and Computation 2, 191–200 (2006)
6. Barth, P.: A Davis-Putnam Enumeration Algorithm for Linear Pseudo-Boolean Optimization. Technical Report MPI-I-95-2-003, Max Plank Institute for Computer Science (1995)
7. Berre, D.L.: SAT4J library, http://www.sat4j.org
8. Biere, A.: PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation 2, 75–97 (2008)
9. Bonet, M.L., Levy, J., Manyà, F.: Resolution for Max-SAT. Artificial Intelligence 171(8-9), 606–618 (2007)
10. Borchers, B., Furman, J.: A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. Journal of Combinatorial Optimization 2, 299–306 (1999)
11. Chai, D., Kuehlmann, A.: A fast pseudo-Boolean constraint solver. In: Design Automation Conference, pp. 830–835 (2003)
12. Coudert, O.: On Solving Covering Problems. In: Design Automation Conference, pp. 197–202 (1996)
13. Darras, S., Dequen, G., Devendeville, L., Li, C.M.: On inconsistent clause-subsets for Max-SAT solving. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 225–240. Springer, Heidelberg (2007)
14. Edmonds, J.: Paths, trees and flowers. Canadian Journal of Mathematics 17, 449–467 (1965)
15. Een, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2, 1–26 (2006)
16. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 252–265. Springer, Heidelberg (2006)
17. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSat: a new weighted Max-SAT solver. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 41–55. Springer, Heidelberg (2007)

18. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSAT: An efficient weighted Max-SAT solver. Journal of Artificial Intelligence Research 31, 1–32 (2008)
19. Larrosa, J., Heras, F., de Givry, S.: A logical approach to efficient Max-SAT solving. Artificial Intelligence 172(2-3), 204–233 (2008)
20. Li, C.M., Manyà, F., Planes, J.: New inference rules for Max-SAT. Journal of Artificial Intelligence Research 30, 321–359 (2007)
21. Liao, S., Devadas, S.: Solving Covering Problems Using LPR-Based Lower Bounds. In: Design Automation Conference, pp. 117–120 (1997)
22. Lin, H., Su, K.: Exploiting inference rules to compute lower bounds for MAX-SAT solving. In: International Joint Conference on Artificial Intelligence, pp. 2334–2339 (2007)
23. Manquinho, V., Marques-Silva, J.: Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. IEEE Transactions on Computer-Aided Design 21(5), 505–516 (2002)
24. Manquinho, V., Marques-Silva, J.: Effective lower bounding techniques for pseudo-boolean optimization. In: Design, Automation and Test in Europe Conference, pp. 660–665 (2005)
25. Marques-Silva, J., Manquinho, V.: Towards more effective unsatisfiability-based maximum satisfiability algorithms. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 225–230. Springer, Heidelberg (2008)
26. Marques-Silva, J., Planes, J.: On using unsatisfiability for solving maximum satisfiability. Computing Research Repository, abs/0712.0097 (December 2007)
27. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Design, Automation and Testing in Europe Conference, pp. 408–413 (2008)
28. Nieuwenhuis, R., Oliveras, A.: On SAT modulo theories and optimization problems. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 156–169. Springer, Heidelberg (2006)
29. Pipatsrisawat, K., Palyan, A., Chavira, M., Choi, A., Darwiche, A.: Solving weighted Max-SAT problems in a reduced search space: A performance analysis. Journal on Satisfiability Boolean Modeling and Computation (JSAT) 4, 191–217 (2008)
30. Ramírez, M., Geffner, H.: Structural relaxations by variable renaming and their compilation for solving MinCostSAT. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 605–619. Springer, Heidelberg (2007)
31. Safarpour, S., Mangassarian, H., Veneris, A., Liffiton, M.H., Sakallah, K.A.: Improved design debugging using maximum satisfiability. In: Formal Methods in Computer-Aided Design (2007)
32. Sheini, H., Sakallah, K.A.: Pueblo: A hybrid pseudo-Boolean SAT solver. Journal on Satisfiability, Boolean Modeling and Computation 2, 165–189 (2006)
33. Warners, J.: A linear-time transformation of linear inequalities into conjunctive normal form. Information Processing Letters 68(2), 63–69 (1998)
34. Xu, H., Rutenbar, R.A., Sakallah, K.A.: sub-SAT: a formulation for relaxed boolean satisfiability with applications in routing. IEEE Transactions on CAD of Integrated Circuits and Systems 22(6), 814–820 (2003)

PaQuBE: Distributed QBF Solving with Advanced Knowledge Sharing

Matthew Lewis[1], Paolo Marin[2], Tobias Schubert[1], Massimo Narizzano[2], Bernd Becker[1], and Enrico Giunchiglia[2]

[1] {lewis,schubert,becker}@informatik.uni-freiburg.de
http://ira.informatik.uni-freiburg.de/
[2] {paolo.marin,massimo.narizzano,enrico.giunchiglia}@unige.it
http://www.star-lab.it/

Abstract. In this paper we present the parallel QBF Solver *PaQuBE*. This new solver leverages the additional computational power that can be exploited from modern computer architectures, from pervasive multicore boxes to clusters and grids, to solve more relevant instances and faster than previous generation solvers. *PaQuBE* extends *QuBE*, its sequential core, by providing a Master/Slave *Message Passing Interface* (MPI) based design that allows it to split the problem up over an arbitrary number of distributed processes. Furthermore, *PaQuBE*'s progressive parallel framework is the first to support advanced knowledge sharing in which solution cubes as well as conflict clauses can be shared. According to the last QBF Evaluation, *QuBE* is the most powerful state-of-the-art QBF Solver. It was able to solve more than twice as many benchmarks as the next best independent solver. Our results here, show that *PaQuBE* provides additional speedup, solving even more instances, faster.

Keywords: Parallel QBF Solving, Message Passing, Master/Slave Architecture, MPI.

1 Introduction

Recently, Boolean Satisfiability (SAT) solvers have become powerful enough to solve many practically relevant problems, and they are currently used in numerous industrial tools for circuit verification. Building apon this success, the research community has begun to consider the more general (but also more complicated) Quantified Boolean Formula (QBF) domain. This allows researchers to encode problems encountered in Black Box or Partial Circuit Verification [1], Bounded Model Checking [2], and AI planning [3] more naturally and compactly than in SAT. However, since QBF problems are generally more difficult (PSPACE-Complete vs. NP-Complete), they require dedicated algorithms and increased computation power to solve relevant instances. In this context, using multi-processor systems and parallel algorithms is a possible and interesting solution.

While many QBF solvers are still based on the DPLL algorithm [4], they have advanced considerably in recent years. For instance, some QBF algorithm

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 509–523, 2009.

specific advances include conflict and solution analysis with non-chronological backtracking [5,6,7,8], and preprocessing [9,10]. Modern QBF solvers must combine all these new ideas into an efficient implementation to be competitive.

Furthermore, single processor performance has also played a large role in the ability of modern QBF solvers to handle relevant problems. For many years clock frequencies, and single core performance increased rapidly. However, current improvements in clock frequency and single core processor performance are slowing. To compensate for this, we have seen the introduction of multi-core and/or multithreaded processors which have resulted in some of the largest jumps in performance potential in recent times. Companies such as INTEL, AMD, SUN, and IBM, now produce CPUs that contain four or more cores. Future QBF solvers must harness this untapped potential if they wish to provide leading edge performance. These new processors, and the introduction of cheap clusters in labs are the main motivation for the development of *PaQuBE*.

The following section will start with a description of the QBF problem, and how sequential and parallel QBF solvers work. Sections 3 and 4 will talk about the design and implementation of *PaQuBE*, and the performance results that were obtained. Finally, Section 5 will conclude this paper with some closing remarks, and discuss future directions we wish to take *PaQuBE*.

2 QBF Problem/Solver Overview

In our context, QBF formulas are defined in Conjunctive Normal Form (CNF). A problem in CNF form would first consist of a variable definition, typically containing multiple alternations of existentially and universally quantified variables. More formally, a *QBF* is an expression of the form:

$$\varphi = Q_1 z_1 Q_2 z_2 \dots Q_n z_n \Phi \qquad (n \geq 0) \tag{1}$$

Here, every Q_i $(1 \leq i \leq n)$ is a quantifier, either existential $\exists$ or universal $\forall$, $z_1, \dots, z_n$ are distinct sets of variables, and Φ is a propositional formula. $Q_1 z_1 \dots Q_n z_n$ is defined as the *prefix*, and Φ, the propositional formula, would contain a set P of clauses. While a *variable* is defined as an element of P, an occurrence of that variable or its negation in a clause is referred to as a *literal*. In the following, the literal $\overline{l}$ is defined as the negative occurrence of varable $|l|$ in P, and l is the positive occurrence. In the following, we also use TRUE and FALSE as abbreviations for the empty conjunction and the empty disjunction, respectively. For example, an entire problem definition might be as follows:

$$\exists x_1 \forall y \exists x_2 \{\{\overline{x}_1 \vee \overline{y} \vee x_2\} \wedge \{\overline{y} \vee \overline{x}_2\} \wedge \{x_2\} \wedge \{x_1 \vee \overline{y}\} \wedge \{y \vee x_2\}\} \tag{2}$$

We say that (1) is in *Conjunctive Normal Form* (CNF) when Φ is a conjunction of *clauses*, where each clause is a disjunction of literals as shown in (2). And that (1) is in *Disjunctive Normal Form* (DNF) when Φ is a disjunction of *cubes*,

where each cube is a conjunction of literals[1]. We use *constraints* when we refer to clauses and cubes indistinctly. Finally, in (1), we define

- the *level of a variable* z_i, to be 1 + the number of alternations $Q_j z_j Q_{j+1} z_{j+1}$ in the prefix with $j \geq i$ and $Q_j \neq Q_{j+1}$;
- the *level of a literal* l, to be the level of $|l|$.
- the *level of the formula* (1), to be the level of z_1.

For example, in (2) x_2 is existential and has level 1, y is universal and has level 2, x_1 is existential and has level 3.

2.1 Sequential QBF Solver

A DPLL based solver would start by reading the formula. Then, using a heuristic, one of the variables in the formula would be chosen and assigned a value (TRUE or FALSE). In the QBF domain, the decision heuristic is restricted to choosing variables on the first quantification level. Only when all the variables on this level are defined, can the heuristic move on to the next level. Once a decision is made, a Boolean Constraint Propagation (BCP) procedure is run. The BCP procedure finds the implications or consequences of that decision. If the BCP procedure completes and no conflicts are found, the decision procedure is run again. However, if a conflict is found, a conflict analysis procedure is run in order to find the reason for the conflict. It would then try to resolve the conflict by backtracking to a previous decision level. If the conflict cannot be resolved, the problem is unsatisfiable. However, if backtracking can resolve the conflict, a conflict clause would also be recorded to prevent the solver from repeating this error in the future.

In case of QBF Solvers like *QuBE* [11] which feature Solution Backjumping, whenever a solution is found, an initial reason can be computed in order to run the above conflict analysis procedure almost symmetrically, thus recording a solution cube. If the solution cannot be resolved, the problem is satisfiable.

Note, while many QBF solvers run in the way described above, there are many details not covered here, and we refer the reader to [7,8,11,12].

2.2 Parallel QBF Solver

In our context, a parallel QBF solver consists of multiple copies of a sequential solver. Each sequential solver (in the total parallel solver) functions in the same manner as described in Section 2.1. However, instead of working on the entire problem, each individual solver is given a small part of the original problem. This is done by dividing the search space into 2 or more disjoint parts. This can be accomplished by selecting a decision variable and telling each solver to search opposite assignments of that variable. This method is referred to as the Guiding Path method in SAT and it was first introduced by PSATO [13]. Normally, in SAT, the chronologically first decision variable is taken as shown in Figure 1. The search space can

[1] Solution cubes are also referred to as solution terms.

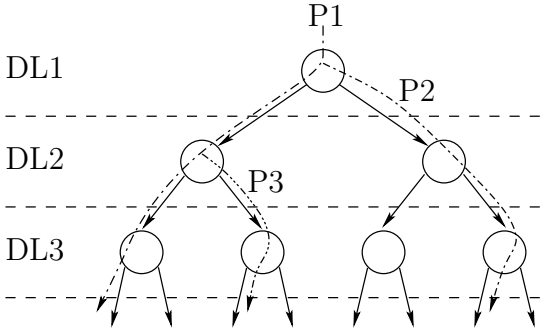


Fig. 1. Guiding Paths

be divided repeatedly in this manner by choosing newer decision variables. However, when using this method to produce subproblems in QBF, a more elaborate mechanism must be in place to keep track of which parts of the search space are currently being searched, and which parts have already been proved satisfiable or unsatisfiable. This is because different clients will not only have different available splitting variables, but these variables could be on different quantification levels.

2.3 Current Sequential QBF Solvers

There are many sequential QBF solvers. Most solvers like QMiraXT [14], QuBE [6], yQuaffle [15], sSolve [16], are in principal based on the DPLL algorithm. Others, like Quantor [17] or Nenofex [18], try to resolve and expand the formula until no universally quantified variables remain. This allows them then to send their remaining, existentially quantified problem to a SAT solver. This works well on many problems, but it can result in an explosion with respect to the size of the formula. On the other hand, solvers like sKizzo [19] do the opposite of Quantor, and use symbolic skolemization to eliminate all the existentially quantified variables in the formula. Some so-called incomplete solvers (e.g. WalkQSAT [20]) are also based on stochastic search methods, and they can be very effective in solving some categories of problems, but are not able to prove the value of unsatisfiable formulas. Finally, in [21,22] a portfolio of solvers is considered, and the best one is selected using machine learning techniques.

While many of these techniques show promise, our focus here is on QuBE, which is a DPLL based algorithm. It would however be interesting to see if algorithms like Quantor's "Resolve and Expand" and sKizzo's "Symbolic Skolemization" could be parallelized. Or even what algorithms like [21,22] could do if they ran a portfolio of algorithms at once, but this is open for future research.

2.4 Previous Parallel QBF Solver Work

The parallelisation of SAT has been studied e.g. in [13,23,24]. There is, however, only one parallel QBF solver that we are aware of, called PQSOLVE [25], that

takes advantage of Message Passing. PQSOLVE was based on the basic DPLL algorithm, without conflict analysis, solution analysis, watched literals, and many other advanced techniques used in QBF solvers today. PQSOLVE also had many limitations with respect to problem splitting. First, PQSOLVE needed to keep track of a complicated list of parent and children nodes that described who donated and received which subproblem. This was required to ensure completeness. Additionally, even on random problems that should be easier to parallelize in a basic DPLL search, idle times for the system with 32 processors were about 16% (increasing to 31% for 128 processors). Lastly, since PQSOLVE did not include any type of conflict or solution analysis procedures, aspects like clause or cube sharing were not even relevant. This is not to say that PQSOLVE was not a novel solver. On the contrary, at the time it was published it was a state-of-the-art solver, but that was almost a decade ago.

Recently, as the need for parallel QBF algorithms has become more apparent, the threaded parallel SAT solver MiraXT [24] was modified so that it could directly handle QBF formulas [14]. QMiraXT was developed to use multicore/multi-CPU workstations. Its tight integration of threads allows significantly more knowledge sharing than an MPI design. QMiraXT also introduced some novel ideas on how to transform ideas from the parallel SAT domain to QBF. For instance, *PaQuBE* uses some of these ideas such as the Single Quantification Level Scheduling (SQLS) subproblem generation algorithm, and knowledge sharing in a QBF context that QMiraXT introduced. *PaQuBE* builds upon these ideas and includes sharing of solution cubes as well. Furthermore, due to its Master/Slave MPI design, *PaQuBE* is far more scalable than QMiraXT allowing it to take advantage of entire clusters or grids.

3 PaQuBE Design Overview

We now present the parallelisation of *QuBE*, resulting in the distributed QBF proving algorithm *PaQuBE*. *QuBE* is a search based QBF Solver that uses lazy data structures for both unit clauses propagation and for pure literals detection [6]. It also features conflict and solution non-chronological backtracking and learning and it is a competitive state of the art solver[2]. Next, the following sections will describe the general properties of our approach, while focusing on the dynamic partitioning of the overall search space and the cooperation between the processes.

3.1 General Properties

PaQuBE has been implemented following a Master/Slave Model, where one process is dedicated to be the master, and $n - 1$ are acting as slaves that actually perform the solving. Here, n represents the total number of processes running on the system. An illustration, using three clients, is given in Figure 2.

[2] Version 6.5. Actually, *QuBE*6.5 is a composition of the preprocessor *sQueezeBF* and the core solver.

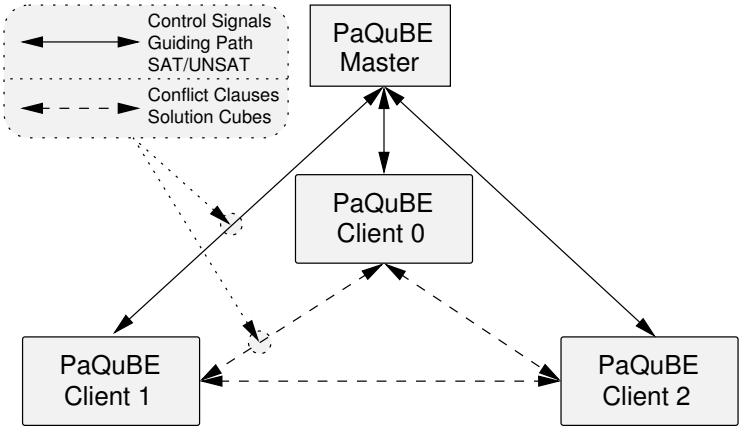


Fig. 2. PaQuBE Design

In our implementation, the master is a central part of the solver and it is required to ensure completeness. In more detail, the role of the master is:

1. Maintain information about the current subproblems and splitting variables.
2. Start the slaves.
3. When there are sleeping slaves waiting for new subproblems, select a working slave and then request a subproblem from it while maintaining the SQLS rules discussed later.
4. Activate sleeping slaves with new subproblems to analyze when new subproblems become available.
5. Stop all the slaves if one of them proves the satisfiability/unsatisfiability of the formula.

Likewise, the role of the slave is:

1. Receive and solve a subproblem, represented as a set of assumptions about the complete problem.
2. Split its subproblem if asked, and then send the new unevaluated part to the master.
3. Share with other slaves some conflict clauses learnt during the search.
4. Compress and share with other slaves some solution terms learnt during the search.
5. Receive and add to the local database part of the conflict clauses and solution terms forwarded by other slaves.

Since the master spends most of its time sleeping, and when working there is at least one inactive slave, it can be run alongside other processes without really needing a dedicated CPU. Indeed, in contrast to many other parallel MPI based SAT solvers, the knowledge sharing mechanism does not involve the master.

The only reason we need a master process is for controlling the SQLS scheduling algorithm. Without a master process, each *PaQuBE* client would need to talk to all other clients before donating a subproblem. This would generate significantly more messages compared to the Master/Slave model used in our approach.

The core solver used in each slave is *QuBE*, tweaked in order to deal with assumptions and to work in a group environment. In particular, the backjumping engine has to treat subproblem assumptions like decision literals when evaluating the reasons for conflicts, i.e. even if they have been assigned at decision level 0 they cannot be resolved out of the formula. We have also added a procedure to correctly set the watched literals in those clauses or terms learnt from other slaves, while also backtracking if possible. Furthermore, during the search each slave must check for messages coming from the master (e.g. requests for subproblems or a notification that the problem has been solved) or a slave (incoming clauses or terms). This check is done regularly after a fixed number of assignments. Whenever a slave checks for messages, learnt constraints are shared with other slaves if selected as suitable under the knowledge sharing mechanism.

The entire communication has been realized using MPICH2 [26], an implementation of the Message Passing Interface standard [27]. According to the Master/Slave model sketched above, all communication tasks are encoded as *messages* and sent/received using `MPI_Send` and `MPI_Recv`, respectively.

3.2 Initialization

At start-up, the slaves read in the preprocessed input formula. The preprocessor *sQueezeBF* is used here, and it is the only sequential part of this parallel solver. Afterwards, slave #1 sends the master a few basic properties of the formula so that it can initialize the SQLS scheduler. In particular, these are the number of variables, clauses, and the number of quantification levels or alternations in the input formula. Then, slave #1 begins the search trying to solve the complete problem, i.e. the given formula without any assumptions. The slaves from #2 to n request and then wait for incoming subproblems to solve. Waiting slaves are put to sleep (using the `MPI_Iprobe` command) so that they do not affect the performance of running slaves.

3.3 Single Quantification Level Scheduling

For parallel QBF, the total search space has to be divided into disjoint fractions. We adopt the dynamic splitting technique called Single Quantification Level Scheduling (SQLS) which was introduced in [14]. SQLS basically divides the search space in a fashion similar to PSATO, using the first decision variable assigned by the decision heuristic. However, because we now have a QBF formula instead of a SAT formula, the master must keep track of more information. In PQSOLVE, this task was quite complicated. SQLS simplifies this, allowing the master to only control the quantification level of the variables being used to generate new subproblems (root variables), while also keeping track of how many slaves are actually running.

In SQLS, the master will first ask for subproblems with a root variable initialized to the first level in the formula. Whenever a slave asks the master for a subproblem, this request will be forwarded to a working slave, requiring that the subproblem must be rooted at the current quantification level. If the first branch done by the inquired slave is not quantified on the correct level, the master tries again with another slave. In case all the variables quantified at this level have already been checked, the master will move on to the next level when there is at most one running process. This poses the only limit over the maximum number of processes that can be run simultaneously. However, in most cases this is not a limiting factor. With only 10 decision variables on the outermost quantification level, we have 2^{10} possible subproblems. Normally, there are 10s, if not 100s of variables on the first quantification level. Lastly, the master can stop all the slaves as soon as one of them finds its subproblem to be unsatisfiable and the current quantification level is even (universally quantified), or satisfiable with an odd quantification level (existentially quantified).

3.4 Knowledge Sharing

As stated above, *PaQuBE* slaves can share both learnt clauses and terms. As learning made SAT/QBF Solvers able to solve real world problems, acquiring clauses derived from solving parts of the search space can help as well [24]. Moreover, it is well known that computing initial reasons for backjumping from a solution (terms or cubes) is far more expensive than the conflict case (see [8] and [12] for more detailed considerations). As a consequence, sharing small and already computed solution terms may speed up the search. In order to save part of the time (latency and transmission time) needed to send these large messages in general, clauses are packed into bundles, and terms are packed and compressed, with the aim of filling without exceeding the capacity of a TCP packet.

The algorithm used for compressing terms works on the assumption that these terms share many literals, in particular those quantified at the highest levels. This is normally true, especially at the beginning of the search. Therefore, if the literals occurring in these terms are sorted according to the prefix order, in every block of terms we can effectively detect and avoid sending the common part of each. Moreover, every literal that may occur in a term (e.g. those bounded by quantification levels from the highest to the lowest universal) are encoded into two-bits. This encoding allows us to communicate that a literal (i) occurs with a positive polarity (01), (ii) a negative polarity (11), or (iii) does not occur in this term (00). The remaining allowable value is used as a marker for the end of the term. Finally, after converting all the selected terms, we put the complete first term into the packet. Then, for the following terms, we only include the term's differing tail, and an offset pointing to where this term starts to differ from the first one. Consider for example the formula below.

$$\varphi = \forall y_1 y_2 \exists x_1 x_2 \forall y_3 \exists x_3 \varphi \tag{3}$$

Excluding the innermost existential variables (those bounded at the lowest quantification level) 5 atoms may occur in a term (because of minimization). Now, let's say a solver learns the following terms:

$$\{y_1, \neg y_2, x_1, y_3\}, \{y_1, \neg y_2, x_1, \neg x_2, \neg y_3\} \tag{4}$$

Their 2−bit encodings are, respectively,

$$\{\texttt{01}|\texttt{11}|\texttt{01}|\textit{00}|\textit{01}\}, \{\texttt{01}|\texttt{11}|\texttt{01}|\textit{11}|\textit{11}\} \tag{5}$$

Only the last 2 literals (highlighted in *italic*) differ. We say: *"the difference begins at the* 4^{th} *position"*. Then, the sent message will be:

$$\{\texttt{01}|\texttt{11}|\texttt{01}|\texttt{00}|\texttt{01}|\mathbf{4}|\texttt{11}|\texttt{11}\} \tag{6}$$

Here, the comparison between terms has been done literal by literal (pairs of bits), but for the sake of efficiency in *PaQuBE* this is done between sets of 16 literals, that are 32 bits long.

Now, when receiving clauses or cubes, slaves only add ones that are either short, conflicting, or producing implications. This eliminates adding many unuseful clauses or terms, while providing a balance between the knowledge sharing and the number of clauses the BCP procedure must evaluate. However, it has a limitation already known from parallel SAT solvers based on message passing: when a slave selects constraints to be shared, is not aware of their usefulness to other slaves. This is because slaves are not aware of other slaves' current status or subproblems. In order to exchange this information and keep it up-to-date would imply either too many messages or too great of latency if updated just before sharing. Being able to select the constraints in this way however, would allow us to share even larger ones more effectively, even if less knowledge in total was shared. This is an interesting approach that we are currently developing.

4 Experimental Results

To evaluate the performance of *PaQuBE* and the effectiveness of our ideas, we have run multiple experiments on a selected pool of fixed-structure instances from $qbflib$ [28]. The benchmarking machine used in this section was a Sun Fire X4440. It contains four Quadcore AMD Opteron 8356 processors. Each processor runs at 2.3 GHz, and is connected to 16 GB of local memory (64 GB in total). This machine runs a 64 bit version of the Linux 2.6.24 kernel, and supports the MPICH 2-1.0.8 library. Also, in our current setup, *PaQuBE*'s knowledge sharing is tuned for this AMD system. *PaQuBE*'s information bundles contain the last 20 learnt conflict clauses or cubes. Each clause is limited to a size that contains $< 15\%$ of the variables within the problem, and cubes having a length of $< 18\%$ of the variables within the formula minus the number on the last existential level (if one exists). These numbers were experimentally determined to perform best.

This AMD system provides significantly more performance for message passing (with respect to latency and throughput) than a distributed system such as

Table 1. PaQuBE: Performance Scaling

Solver	#CC/s	#SC/s	#SP	#PS	Time	Speedup
QuBE 1P	0	0	1.00	227	63052.91	1.00
PaQuBE 2P	38.10	40.40	3.47	242	48387.99	1.34
PaQuBE 4P	79.45	73.78	8.86	262	33592.96	1.89
PaQuBE 8P	109.87	102.92	18.05	266	31797.71	1.98
PaQuBE 16Pns	—	—	40.71	272	31805.73	1.98
PaQuBE 16P	157.09	137.99	38.21	274	27857.26	2.26

a grid that is connected by Ethernet. On a larger cluster, *PaQuBE*'s knowledge sharing would have to be scaled down accordingly with the bandwidth available. However, on our AMD system, we do have the issue that multiple cores on one processor must share that processor's memory bus. On our Sun Fire X4440 server, each of the four AMD Opteron 8356 processors has its own memory bus, so we can run 4 *PaQuBE* clients (one client on each processor) before we run into memory bus contention issues. When running the $16P$ case, the 4 cores on each processor must share the processor's memory bus. This unfortunately affects the scaling of the algorithm in the $> 4P$ cases. Fortunately, *PaQuBE*'s MPI architecture can seamlessly adapt to both multicore workstations and grids, allowing it to leverage the advantages that each type of parallel system provides.

In Table 1 and 2 we compare the performance of the sequential solver *QuBE* to that of *PaQuBE* running on 2, 4, 8, and 16 slaves (marked as $2P/4P/8P/16P$). To do this, we selected the benchmarks from *qbflib* for which *QuBE*, the sequential solver, needed between 10 and 600 seconds. We then added the next few incrementally harder benchmarks from each family to see if *PaQuBE* could also solve more instances. In total, over 20 different benchmark families were tested. These families are shown in column one of Table 2. For benchmarking, each version of the solver was run once on the complete list of benchmarks. Each version was given 600 seconds to solve each instance, and in all the tables, the columns titled $\#CC/s$ and $\#SC/s$ represent Conflict Clauses and Solution Cubes shared per second, while $\#SP$ represents the number of subproblems that were generated on average.

First, Table 1 clearly shows the advantage of running *PaQuBE* on more processors. Column 6, labelled $Time$, shows the real world time used to solve all 283 instances (unsolved instances are included with the timeout value of 600 seconds), and shows that *PaQuBE* provides good speedup from $1P$ to $16P$, in terms of time as well as in terms of the number of problems solved ($\#PS$). The best performance scaling is from $1P$ to $4P$. As we move on to the $8P$ and $16P$ cases, performance increases, but it does not scale as nicely as the $1P$ to $4P$ case. This can be seen in both the speedup column, as well as when comparing how much information was exchanged between the slaves. For example, while the number of $\#CC/s$ and $\#SC/s$ scale linearly from $2P$ to $4P$, resulting in almost exactly twice the amount of information exchanged, the difference from $4P$ to $16P$ case is only double even though the number of processors were quadrupled.

Table 2. PaQuBE: Benchmark Family Performance

Family	#Inst.	2P ×	4P ×	16P ×	16P Solver			
					#SP	#CC/s	#SC/s	CPU U.%
Abduction	13	1.32	9.89	7.7	33.4	423.3	0	94
BMC	12	1.18	1.07	1.35	225.8	1192.02	15.23	99
Cond. Pl.	2	2.12	2.71	2.46	51	112.61	0	96
counter	1	1.5	1.73	1.04	128	177.22	5.48	99
Ev-Pr-*-lg	7	1.01	11.86	11.97	9	27.89	11.28	88
FPGA_PFS	1	1.96	8.71	22.13	44	168.41	0	95
irqlkeapcite	1	2.57	5.1	6.04	34	0	0	14
k_*_n	16	1.14	1.16	2.73	13.3	90.76	0.72	96
k_*_p	18	1.25	1.57	1.54	35.7	36.99	13.98	95
katz	2	10.67	56.14	55.65	35.5	371.01	32.72	94
logn	1	1.37	1.79	1.58	21	74.48	0	95
sakallah	43	1.23	1.51	1.51	48.5	3.2	519.04	98
Scholl-Becker	7	1.33	1.4	1.44	31.7	254.68	0	98
Sorting Netw.	15	0.6	1.19	3.61	42.3	478.2	599.33	83
Szymanski	4	1.13	1.7	1.66	19	0	33.14	76
terminator	21	1	2.51	192.8	14.6	0.8	156.46	63
tipfixdiameter	32	2.36	2.69	3.11	34	140.6	114.12	94
tipfixpoint	79	1.36	1.71	1.44	21.7	91.63	0.61	99
TOILET	5	4.78	9.71	9.74	17.6	20.55	0	95
wmiforward	3	4.49	7.8	2.53	44.3	0	9.2	23
Total	283	1.52	3.01	3.06	38.2	157.09	137.99	97

This is mainly due to bus contention on each processor as was briefly described earlier. In any case, even with the current AMD architecture, we are still able to increase performance all the way up to $16P$. The largest increase however, was not in time, but in problems solved. With $16P$ we solved 47 more instance than the $1P$ case. Even more exciting, was the fact that we solved 13 problems that had not been solved by any solver at the last QBF competition.

Next, as was also shown in [14], the low number of subproblems generated means that the SQLS's limitations do not limit the performance of the solver as there is rarely a need to generate lots of subproblems. Furthermore, the column labeled *CPU U.%* from Table 2, which shows the CPU utilisation time, adds additional support to this argument as the average CPU utilisation was 97%. Simply put, this means on average, only 3% of the time was a CPU idle, waiting for a subproblem. Table 1 also includes $PaQuBE\ 16Pns$. This is *PaQuBE* with knowledge sharing disabled, demonstrating the impact of knowledge sharing. As can be seen, knowledge sharing improves the $16P$ case by almost 16%, even though we have an overhead of sending $2820k$ messages compared to the no sharing $16P$ case.

Next, Table 2 takes a closer look at each benchmark family. The first two columns contain the benchmark family name (*Family*) and how many instances from that family were included (*#Inst.*). This table then shows the speedup for the $2P/4P/8P/16P$ cases. For the $16P$ case it also includes the number of

subproblems generated, and conflict clauses and solution cubes shared on benchmark family basis. This is interesting as *PaQuBE*'s scaling performance on different benchmark classes is substantial. On families such as `katz`, `Ev-Pr-*-lg`, and `Abduction` the performance is excellent, but on families such as `BMC` and `k_*_n` there is no performance increase at all. There are two main reasons for poor performance on certain benchmarks. First, there are benchmarks that for instance use existentially quantified variables to produce subproblems, but in which all subproblems are satisfiable. This results in each *PaQuBE* client needlessly searching a satisfiable subproblem, when only one satisfiable subproblem needs to be searched. Thankfully, with intelligent conflict clause and solution cube sharing, each *PaQuBE* client can still learn from one another, thus minimising this redundant work. Secondly, on some benchmarks, the solver is mostly on decision level 0 during the evaluation (e.g. `irqlkeapcite` and `wmiforward`). This means that no subproblems can be generated, resulting in many processors being idle. The low CPU utilisation then results in lower parallel performance. Fortunately, these are the only two benchmark families that suffer from this. Regarding the number of conflict clauses and solution cubes shared, Table 2 shows that moderate sharing provides the best speedup. Problems that share too much or not enough don't scale as well. However, on the vast majority of benchmarks, good speedup is obtained.

With respect to benchmarks like `terminator` and `katz`, in which we achieve super linear speedup, this is basically attributed to the fact that one of the 16 clients received a subproblem that produced a conflict that showed that the entire problem was unsatisfiable. This again is an advantage of a parallel solver. Decision heuristics are not perfect, and by adding more clients, we have a better chance of sending the solver to a more fruitful part of the search space.

Table 3. PaQuBE: Benchmark Family Performance

Category		#		2P		4P		8P		16Pns		16P	
SAT		195		1.35		1.89		1.98		1.86		2.01	
∃	∀	185	10	1.39	0.77	1.90	1.54	1.94	3.75	1.86	1.85	1.97	3.92
UNSAT		88		1.22		1.91		2.00		2.27		2.96	
∃	∀	59	29	1.50	1.02	1.69	2.21	1.71	2.45	1.28	14.88	1.73	14.18

Table 3 shows the impact of our approach on instances that are satisfiable (SAT) or unsatisfiable ($UNSAT$). It also divides each of these results into two separate categories. Mainly, problems that start with ∃ or ∀ quantification levels. These variables are the most likely to be used as splitting variables. It also shows how many problems belong to each set (labeled #). This table shows that while $UNSAT$ problems scale better than SAT, problems that start with universally quantified variables scale even better (both in the SAT and $UNSAT$ case). Also, it can be seen when comparing $16Pns$ to $16P$, sharing seems to help on all types of instances.

Lastly, as can be seen from all the results presented here, good speedup can be obtained on QBF problems using parallel algorithms. However, this general

statement is benchmark related as certain problems benefit more than other from this parallel approach.

5 Conclusion and Future Work

In this paper we introduced the parallel QBF solver *PaQuBE*. It is based on the state-of-the-art QBF solver *QuBE*, which according to the last QBF competition is significantly faster than other sequential solvers. The new parallel solver *PaQuBE*, not only matches the performance of *QuBE*, but solves 47 more benchmarks and reduces the solving time of families by over 3 times when running with 16 clients, including 13 instances never before solved at the QBF competition, making it the fastest general purpose QBF solver we know of. Lastly, because of its flexible architecture, it can easily scale from 1P to 16P to take full advantage to today's and tomorrow's multicore processors.

In the future, we plan to push PaQuBE's scaling limits by testing it on an even larger cluster, currently being installed at the University of Genova. This cluster will contain multiple, multicore IBM servers connected by an Infiniband network (20Gb/s) with over 40 processors in total when installed later this year. Using the PaQuBE work presented here as a solid foundation, we will hopefully be able to solve larger and even more interesting problems in the near future.

Acknowledgment

The authors would like to thank the German DAAD and the Italian AIT for their support (Vigoni). Furthermore, this work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

References

1. Herbstritt, M., Becker, B.: On Combining 01X-Logic and QBF. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 531–538. Springer, Heidelberg (2007)
2. Dershowitz, N., Hanna, Z., Katz, J.: Bounded model checking with QBF. In: Bacchus, F., Walsh, T. (eds.) SAT 2005, vol. 3569, pp. 408–414. Springer, Heidelberg (2005)
3. Rintanen, J.: Constructing conditional plans by a theorem prover. Journal of Artificial Intelligence Research 10, 323–352 (1999)
4. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem proving. Communication of ACM 5(7), 394–397 (1962)
5. Gent, I.P., Rowley, A.G.: Solution learning and solution directed backjumping revisited. Technical Report APES-80-2004, APES Research Group (February 2004), `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`

6. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/term resolution and learning in the evaluation of quantified Boolean formulas. Journal of Artificial Intelligence Research (JAIR) 26, 371–416 (2006)
7. Kleine-Büning, H., Karpinski, M., Flögel, A.: Resolution for quantified Boolean formulas. Information and Computation 117(1), 12–18 (1995)
8. Zhang, L., Malik, S.: Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 200–215. Springer, Heidelberg (2002)
9. Giunchiglia, E., Marin, P., Narizzano, M.: An effective preprocessor for QBF prereasoning (2008)
10. Samulowitz, H., Davies, J., Bacchus, F.: Preprocessing QBF. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 514–529. Springer, Heidelberg (2006)
11. Giunchiglia, E., Narizzano, M., Tacchella, A.: QuBE++: An efficient QBF solver. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 201–213. Springer, Heidelberg (2004)
12. Giunchiglia, E., Marin, P., Narizzano, M.: 24. In: Reasoning with Quantified Boolean Formulas. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 761–780. IOS Press, Amsterdam (2009)
13. Zhang, H., Bonacina, M.P., Hsiang, J.: Psato: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. 21(4-6), 543–560 (1996)
14. Lewis, M., Schubert, T., Becker, B.: QMiraXT – A Multithreaded QBF Solver. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (January 2009)
15. Yu, Y., Malik, S.: Validating the result of a quantified boolean formula (QBF) solver: theory and practice. In: Tang, T.A. (ed.) ASP-DAC, pp. 1047–1051. ACM Press, New York (2005)
16. Feldmann, R., Monien, B., Schamberger, S.: A distributed algorithm to evaluate Quantified Boolean Formulae. In: Proc. AAAI (2000)
17. Biere, A.: Resolve and expand. In: Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 59–70. Springer, Heidelberg (2005)
18. Lonsing, F., Biere, A.: Nenofex: Expanding nnf for QBF solving. In: Büning, H.K., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 196–210. Springer, Heidelberg (2008)
19. Benedetti, M.: sKizzo: A suite to evaluate and certify QBFs. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 369–376. Springer, Heidelberg (2005)
20. Gent, I.P., Hoos, H.H., Rowley, A.G.D., Smyth, K.: Using stochastic local search to solve quantified Boolean formulae. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 348–362. Springer, Heidelberg (2003)
21. Pulina, L., Tacchella, A.: A multi-engine solver for quantified Boolean formulas. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 574–589. Springer, Heidelberg (2007)
22. Samulowitz, H., Memisevic, R.: Learning to solve QBF. In: Proc. AAAI, pp. 255–260 (2007)
23. Chrabakh, W., Wolski, R.: Gridsat: A chaff-based distributed sat solver for the grid. In: SC 2003: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Washington, DC, USA, p. 37. IEEE Computer Society, Los Alamitos (2003)
24. Lewis, M., Schubert, T., Becker, B.: Multithreaded SAT Solving. In: ASP Design Automation Conf., Yokohama, Japan (January 2007)

25. Feldmann, R., Monien, B., Schamberger, S.: A distributed algorithm to evaluate Quantified Boolean Formulae. In: Proceedings of the 7th Conference on Artificial Intelligence (AAAI 2000) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI 2000), July 30-3, pp. 285–290. AAAI Press, Menlo Park (2000)
26. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22(6), 789–828 (1996)
27. Snir, M., Otto, S., Walker, D., Dongarra, J., Huss-Lederman, S.: MPI: The Complete Reference. MIT Press, Cambridge (1995)
28. Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantified Boolean Formulas satisfiability library (QBFLIB) (2001), `www.qbflib.org`

c-sat: A Parallel SAT Solver for Clusters

Kei Ohmura and Kazunori Ueda

Dept. of Computer Science and Engineering, Waseda University
3-4-1, Okubo, Shinjuku-ku, Tokyo 169-8555, Japan

Abstract. Parallelizing modern SAT solvers for clusters such as Beowulf is an important challenge both in terms of performance scalability and stability. This paper describes a SAT Solver c-sat, a parallelization of MiniSat using MPI. It employs a layered master-worker architecture, where the masters handle lemma exchange, deletion of redundant lemmas and the dynamic partitioning of search trees, while the workers do search using different decision heuristics and random number seeds. With careful tuning, c-sat showed good speedup over MiniSat with reasonably small communication overhead on various clusters. On an eight-node cluster with two Dual-Core Opterons on each node (32 PEs), c-sat ran at least 23 times faster than MiniSat using 31 PEs (geometric mean; at least 31 times for satisfiable problems) for 189 large-scale problems from SAT Competition and two SAT-Races.

1 Introduction

Most modern solvers for propositional satisfiability problems (SAT) employ DPLL algorithms, non-chronological backtracking, conflict-driven learning and restart [14]. The cutting-edge performance has been obtained by careful design decisions in data representation (e.g., two-literal watching) and heuristics (on decision variables, restarts, etc.). We believe that a high-performance parallel solver for tens of processing elements should *both* (i) exploit ideas from fast sequential algorithms *and* (ii) achieve good parallel speedup. There have been numerous proposals of parallel solvers (e.g., [1] [4] [5] [6] [8] [9] [13]; see [11] for a survey), but most of them focused on individual techniques evaluated using rather limited classes of benchmark problems and/or on multicore or share-memory platforms. Comprehensive performance study on larger-scale parallel computing environments has not been reported yet for award-winning solvers. Because of the heavy-tailed behavior and enormous variance in the characteristics of individual problems, a small number of benchmark problems (which may well exhibit drastic superlinear speedup) may not represent other problems we have at hand.

This paper reports our SAT solver c-sat, a parallelization of MiniSat using MPI, designed to run efficiently on clusters with tens of processing elements (PEs) connected by Ethernet. Computing environments of this scale are of great importance because they are significantly more powerful than current single multicore machines and can still be found, configured, or assembled easily. The main objective and contribution of this work is to show that careful parallelization of the state-of-the-art SAT solver attains good scalability with reasonably

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 524–537, 2009.

small communication overhead on various clusters. Variance of performance in each run is another important concern in parallel SAT solving, but c-sat showed reasonable performance stability.

2 Parallel SAT Solver: c-sat

We describe the basic design and implementation of c-sat. We chose MiniSat [3] (v1.14.1) as the base of c-sat since it was the fastest open-source solver available when the present work started. It is now a classic solver, but in our evaluation, its performance is quite competitive with that of MiniSat 2.0, the latest open-source version. c-sat is written in C (3500 lines) and uses MPI because of its availability on diverse platforms.

2.1 Layered Master-Worker Architecture

The solver c-sat employs both cooperative parallelism (dynamic partitioning of search space based on work stealing) and competitive parallelism (search of the same search space using different heuristics and parameters), both of which are empowered by lemma exchange. The combination was studied also in [2], but to support the efficient exchange of lemmas and subtrees, c-sat employs a three-tiered master-worker model (Fig. 1) Each worker maintains its own clause database and works on subtrees received from its master, exchanging lemmas with the master. The principle of c-sat's master-worker communication is that it is always initiated by each *worker* at its own convenience. Another key decision is to have a grandmaster layer to reduce communication bottleneck and ensure scalability, based on our initial experiences with the two-tiered architecture that worked poorly on clusters with tens of PEs. The grandmaster deals with lemma exchange only, while masters handle partitioning as well. The frequency and the amount of global communication can be tuned depending on the platform and independently of local communication under individual masters.

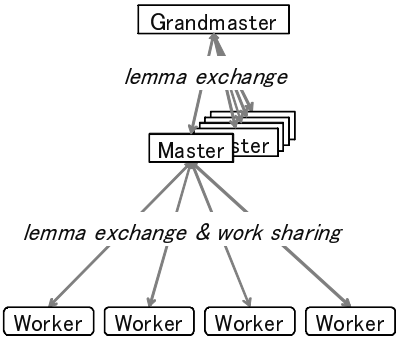


Fig. 1. The three-layer architecture of c-sat

2.2 Decision Heuristics

Each PE chooses decision variables (variables whose values are guessed next) using either of the two versions of VSIDS (Variable State Independent Decaying Sum) heuristics assigned to it: One is the MiniSat version that assigns an activity to each *variable* and selects a variable with the highest activity, where variables contributing to recent conflicts are considered active. It also chooses variables randomly with probability 0.02 to cope with the heavy-tailed behavior. The other version assigns an activity to each *literal* as in the original VSIDS and makes the highest-activity literal false, and is again combined with the 2% random choice. Preliminary experiments showed that the latter version works at least as well as the former, which implies that using both will contribute to performance and stability. Different workers are given different seeds for the 2% random choices to ensure diversity.

Hereafter we refer to the first heuristics as VSIDS and the second as LIT.

2.3 Dynamic Partitioning of Search Trees

Workers belonging to the same master work on different parts of a search tree using the guiding path technique [13]. To minimize communication and housekeeping overhead, a worker communicates the values of decision variables only. A worker receiving a path replays unit propagation to reach the branching point. The replay is justified by the highly optimized unit propagation. The length of guiding paths communicated is limited in order to avoid excessive replaying. We made sure that the limit value 5 is large enough to keep all workers busy.

Each worker employs non-uniform heuristics to explore different parts of a search tree, since they may be engaged in overlapping pieces of work. Upon restart of the first worker, the master removes those paths that have become irrelevant, while the other workers keep working until they send lemmas and are notified of restart in exchange.

2.4 Lemma Exchange

Exchanging lemmas learned by individual workers may drastically reduce the search space. To minimize the rediscovering of lemmas, lemmas should be exchanged frequently. However, the choice and the amount of lemmas exchanged and the frequency of exchange should be carefully designed to balance the utility of lemmas and communication overhead. Lemmas could be chosen based on their activities or their lengths. In the latter approach, typical effect of the limit length on performance is given in Fig. 2. The figure shows the existence of the appropriate "zone" of the limit value; too much lemma exchange would increase both communication and internal processing such as unit propagation. However, the appropriate zone depends on individual problems: Figure 3 shows the average length of lemmas learned in the course of search, where the x-axis is the number of restarts triggered. The figure exhibits enormous difference in the

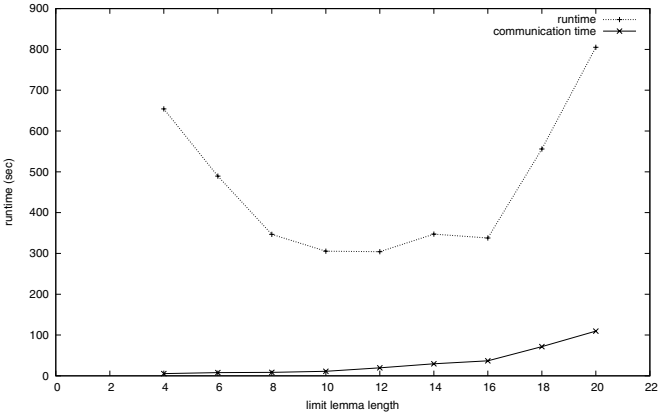


Fig. 2. Maximal length of exchanged lemma vs. execution time and communication time (goldb_heqc_dalumul.cnf (unsatisfiable), 31 PEs, Cluster A of Table 1)

length of lemmas and its change in the course of execution for different problems. Thus we have decided to adaptively determine the limit length based on the minimum and the average lengths of unexchanged lemmas. The limit length takes the number of workers into account as well.

In c-sat, lemma exchange works as follows. Each worker sends lemmas to its master whenever it learns the next 100 new lemmas. The master, in exchange, sends the other workers' lemmas to the worker, and at the same time it exchanges

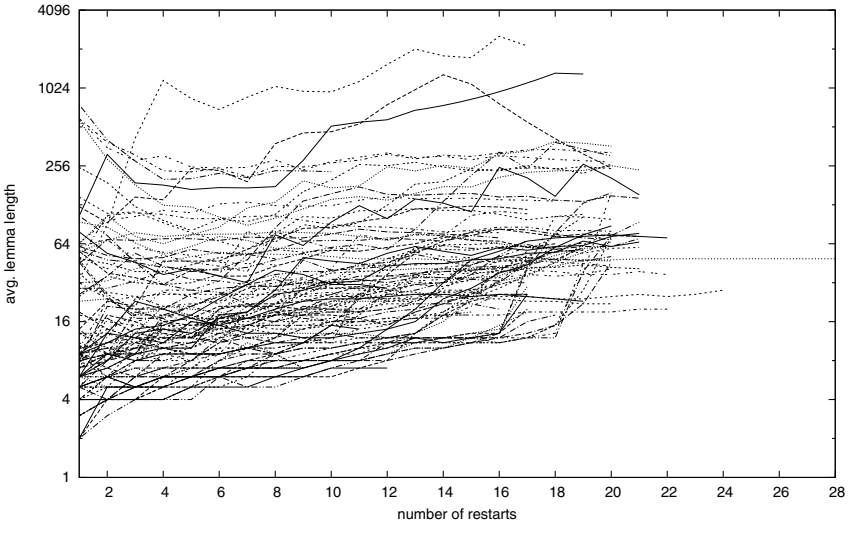


Fig. 3. Change of the length of learned lemmas in 100 different problems from SAT-Race 2006

unexchanged lemmas with the grandmaster. The worker checks the received lemmas for their satisfiability and if a conflicting clause is found, it backtracks to the level at which the clause becomes a unit clause. Each master and the grandmaster may remove lemmas that have been sent to all their partners.

Lemmas are thus distributed to other workers via its master and possibly via its grandmaster. A worker may learn lemmas that are the same as, or subsumed by, lemmas learned by others. The chance of learning redundant lemmas will increase as the number of workers increases. Since the detection and the removal of redundant lemmas are costly operations, only the grandmaster tries to reduce redundancy. Whenever new lemmas are sent from a master, the grandmaster uses the technique of [12] to check for their redundancy with respect to existing lemmas before adding them to the clause database.

3 Experimental Results

We made extensive experiments on c-sat using two clusters listed in Table 1, of which Cluster A was used for full evaluation, while Cluster B was for public use and we were able to use up to 31 nodes and less total execution time.

We tested a total of 286 problems: 100 problems from SAT-Race 2006, 116 problems chosen from SAT 2007 Competition, and 70 problems from SAT-Race 2008. The problems from SAT 2007 Competition were those from the Industrial Track whose sequential execution time exceeded 120 seconds. Problems inherited from past contests have been eliminated from the problem sets of the 2007 and 2008 editions.

Regarding the configuration of the masters-worker model, preliminary evaluation on Cluster A showed good performance on average when each master managed five workers. Accordingly, the main evaluation on Cluster A was conducted using one grandmaster, five masters and 25 workers (31 PEs in total).

3.1 Preliminary Evaluation

Before conducting the main evaluation, we made several preliminary experiments to evaluate our design choices using Cluster A (but possibly using less PEs) on

Table 1. Clusters used for experiments

cluster	CPU/node	nodes	PEs	memory	network
A	2x Dual-Core AMD Opteron 2.0GHz	8	32	4GB/node	GigE
B	1x Intel Core2 Duo 2.13GHz	58	116	4GB/node	GigE

cluster	OS	C compiler	MPI
A	CentOS 4.4	GCC 3.4.6	MPICH2-1.0.8
B	Debian 4.0	GCC 4.1.2	MPICH2-1.0.8

Table 2. Number of problems solved and the total execution time under VSIDS

random seed	±0	−2	−1	+1	+2	Min.
# solved (SAT)	27	30	28	27	29	37
# solved (UNSAT)	36	35	36	38	38	38
# solved	63	65	64	65	67	75
total runtime (SAT)	26271s	23167s	25785s	24687s	22747s	14368s
total runtime (UNSAT)	32065s	31021s	31460s	32099s	30367s	28579s
total runtime	58336s	54188s	57245s	56785s	53115s	42947s

Table 3. Number of problems solved and the total execution time under LIT

random seed	±0	−2	−1	+1	+2	Min.
# solved (SAT)	34	28	26	29	30	37
# solved (UNSAT)	39	38	39	40	39	41
# solved	73	66	65	69	69	78
total runtime (SAT)	20559s	22796s	26518s	24993s	26868s	14409s
total runtime (UNSAT)	30173s	30594s	31170s	29710s	31105s	27480s
total runtime	50732s	53390s	57688s	54703s	57973s	41889s

100 problems from SAT-Race 2006. Due to the variance in the timing of interprocess communication, parallel solvers can exhibit erratic performance behavior. Therefore we ran each problem three times, and regarded those problems solved in *all* runs as "solved" within the time limit.

Decision Heuristics. The variance of sequential execution time due to different heuristics and random seeds is shown in Table 2 (VSIDS) and 3 (LIT). Each table shows the performance with five different random seeds ("±0" standing for the original seed). The rightmost column "Min." chooses the fastest average case of the five seeds for each problem. The "# solved ((UN)SAT)" rows show the numbers of (un)satisfiable problems solved within 1200 seconds, respectively. We counted the execution time of unsolved problems as 1200 seconds here.

The "−2" to "+2" columns reveal that comparing the two heuristics under the default seeds ("±0") doesn't make much sense. The rightmost columns show that simply running five sequential SAT solvers in parallel using different seeds will increase the number of solved problems.

Lemma Exchange. We next evaluated the effect of lemma exchange using a version of c-sat with one master and five workers (6 PEs; no grandmaster). Table 4 repeats the numbers from previous tables (VSIDS and LIT without lemma exchange), and adds the numbers from c-sat with different heuristics. The columns VSIDS+LIT show the results with two workers using VSIDS and three using LIT. The table shows a significant contribution of lemma exchange.

Table 4. The effect of lemma exchange with 6 PEs

heuristics	VSIDS	VSIDS	LIT	LIT	VSIDS+LIT	VSIDS+LIT
lemma exchange	no	yes	no	yes	no	yes
# solved (SAT)	37	38	37	37	39	40
# solved (UNSAT)	38	41	41	45	42	46
# solved	75	79	78	82	81	86
total runtime (SAT)	14368s	13896s	14409s	13569s	17879s	14289s
total runtime (UNSAT)	28579s	24090s	27480s	20490s	23399s	20499s
total runtime	42947s	37986s	41888s	34059s	41278s	34788s

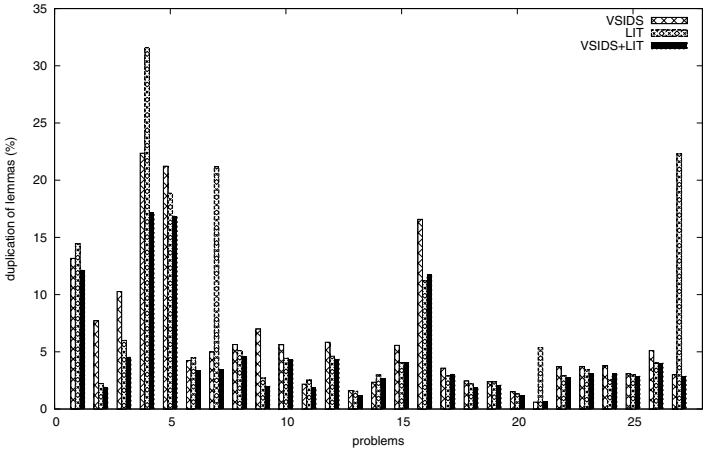


Fig. 4. Proportion of duplicate lemmas in all learned lemmas (for selected problems)

With lemma exchange, VSIDS solved four problems not solved by LIT, and LIT solved six problems not solved by VSIDS. Of those problems, VSIDS+LIT solved eight problems and demonstrated the effect of performance stabilization (recall that the numbers of solved problems in the table count consistently successful runs only). The performance of VSIDS+LIT for individual problems lied between that of VSIDS and that of LIT rather consistently. This was not specific to the particular solver configuration (6 PEs); we observed the same phenomenon using 13 PEs (one grandmaster, two masters and ten workers). We also observed that combining different heuristics may not improve the performance; for instance the combination of VSIDS and random heuristics caused performance degradation for most problems, demonstrating the importance of combining different *and* competitive heuristics.

The effect of combining different heuristics on the reduction of duplicate lemmas is shown in Fig. 4. Positive effect was observed for almost all problems, and duplication was well controlled just by employing different seeds.

Table 5. Cooperative vs. competitive parallelism

dynamic partitioning	no (5 PEs)	yes (5 PEs)	no (13 PEs)	yes (13 PEs)
# solved (SAT)	40	40	40	41
# solved (UNSAT)	46	47	52	52
# solved	86	87	92	93
total runtime (SAT)	14289s	11495s	9195s	8288s
total runtime (UNSAT)	20499s	18367s	13718s	12909s
total runtime	34788s	29862s	22913s	21198s

We also measured the effect of eliminating redundant lemmas using 13 PEs and observed 14% speedup for satisfiable problems. The speedup was rather small for unsatisfiable problems, though.

Finally, we note some statistics of lemma exchange made by the "winning" worker that found a solution first or nonexistence of a solution first. For the SAT-Race 2006 problems, lemma exchange took place 6.8 times per second, the winning worker received 340 clauses per exchange, and the size of clauses exchanged was 5.5 literals, all on average. The total number of lemmas received was 13% of the problem size (geometric mean), though there was enormous variance in this ratio.

Dynamic Partitioning of Search Trees. Comparison between cooperative parallelism (dynamic partitioning) and competitive parallelism with lemma exchange was made using two configurations: 6 PEs (one master) and 13 PEs (two masters) (Table 5). Solvers with dynamic partitioning performed consistently better on clusters with this scale. However, the speedup could be marginal on machines with more communication overhead, and dynamic partitioning is best treated as an option that can be switched off.

3.2 Main Evaluation and Comparison with Sequential SAT

Given the encouraging results of the preliminary evaluation, we made extensive evaluation of c-sat using 31 PEs of Cluster A (one grandmaster, five masters and 25 workers) and the 286 problems mentioned in the beginning of this section. To allow fair comparison with the original MiniSat, parameters including the interval of restarts were kept unchanged. Because an important objective of parallel SAT is to solve large problems, the time limit of each problem was extended to 7200 seconds.

The results are shown in Table 6. The two numbers in the c-sat column show the number of problems solved by *at least one of three runs* (left) and the number of problems solved by *all three runs* (right). Total runtime means average execution time taken to solve all problems solved by all three runs of c-sat. For those problems solved only by c-sat, we re-ran MiniSat for up to 28800 seconds to calculate a better lower bound of the total sequential execution time.

All problems solved by MiniSat were solved by c-sat. Of 247 problems solved by either run of c-sat, 243 were solved by all of the three runs. The small

Table 6. Number of problems solved within 7200s and the total execution time

	MiniSat	c-sat (31 PEs)
# solved (SAT)	80	106 / 103
# solved (UNSAT)	106	141 / 140
# solved	186	247 / 243
total runtime (SAT)	>564691s	21039s
total runtime (UNSAT)	>712627s	51978s
total	>1277318s	73016s

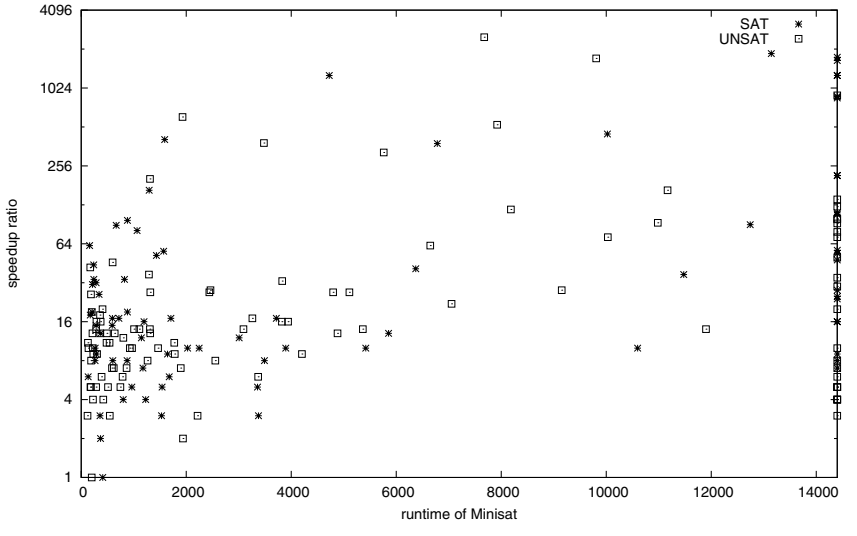


Fig. 5. Performance ratio between c-sat and MiniSat

difference between the two numbers shows the stability effect of using many PEs; with smaller clusters and SMPs, we have experienced much more variance.

Figure 5 shows the performance ratio between c-sat and the original MiniSat, where the x-axis shows the execution time of MiniSat. The figure plots 189 problems whose sequential execution time exceeded 120 seconds. The samples on the $x = 14400$ lines are those whose sequential time exceeded 14400 seconds. The figure shows large speedup for both satisfiable and unsatisfiable problems, particularly for large-scale problems. Superlinear speedup was quite common. The geometric means of the speedup ratio are >31 times for satisfiable problems and >19 times for unsatisfiable problems. Note that arithmetic means, which are inappropriate measures, would be much higher.

Figures 6 and 7 show the stability of execution time for each problem whose c-sat execution time exceeded 120 seconds. Each bar shows the lowest, the average and the highest values of three runs. While the execution time of unsatisfiable problems was quite stable (the higher variance observed for some of heavy problems

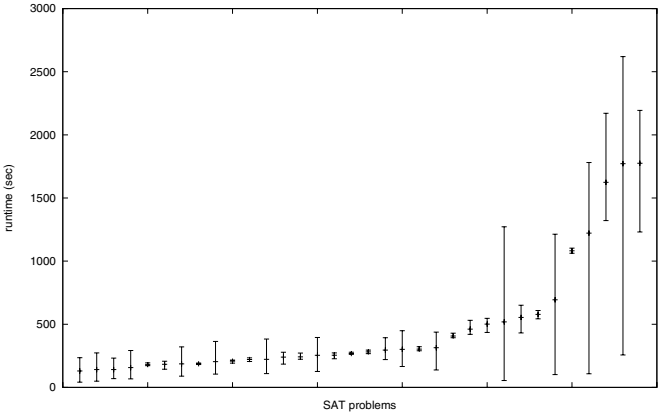


Fig. 6. Variance of execution time (SAT)

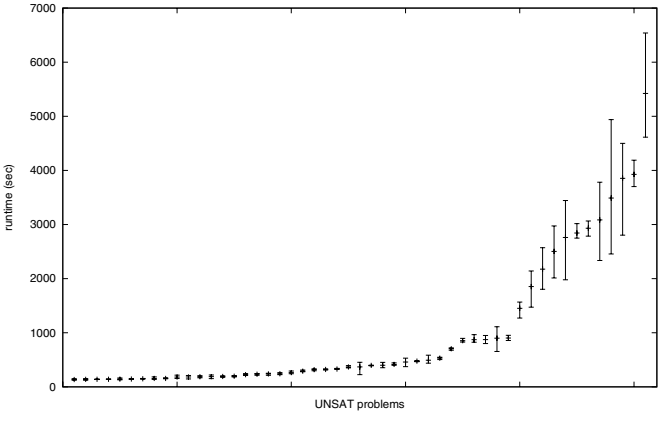


Fig. 7. Variance of execution time (UNSAT)

may possibly be due to the underlying platform rather than the problem itself), the execution time of satisfiable problems had much more variance. This is because the scheduling of inter-PE communication affects the timing of lemma exchange. Taming the performance variance of satisfiable problems seems to have intrinsic difficulty; the only possible solution would be to make more use of randomization, namely more restarts and more competitive parallelism.

We also evaluated communication overhead. Figure 8 shows how much of the wall-clock execution time of the "winning" worker that found a solution first (or nonexistence of a solution first) was spent for lemma exchange. Smaller problems had higher communication overhead in general, but even including them, the average overhead was 9.8%. Communication overhead was almost negligible in solving large problems using 31 PEs. We also measured the communication overhead of dynamic partitioning, but it was even smaller ($< 1\%$).

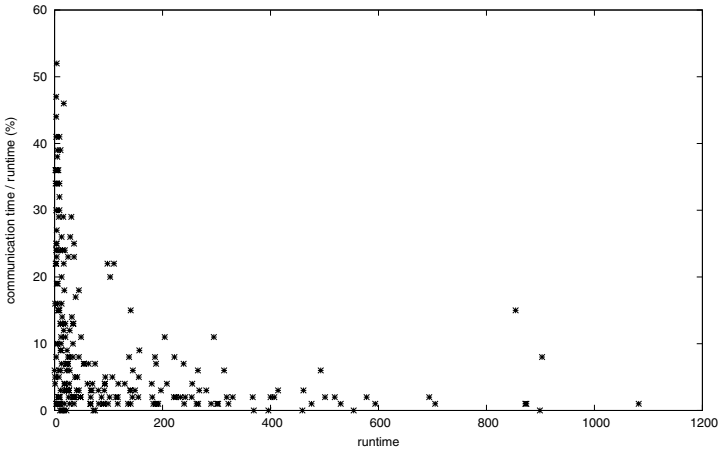


Fig. 8. Time spent on lemma exchange by the winning worker

3.3 Effect of Preprocessing and Comparison with Parallel SAT

The evaluation of the previous section deliberately excluded preprocessing (CNF minimization) to focus on the parallel speedup of the core algorithm. Nevertheless, the effect of preprocessing on parallel performance is of great concern, and whether c-sat outperforms other parallel solvers would be another point of interest. This section outlines our additional evaluation, in which the time limit was set to 1200s for each problem.

Firstly, we built minimized CNF files for all the 286 problems using the preprocessor of MiniSat 2.0, and tested them using MiniSat 1.14 and c-sat (31 PEs). Preprocessing improved the execution time, but the improvement varied widely among individual problems. The parallel speedup was not reduced by preprocessing: for the 149 preprocessed problems whose MiniSat execution time exceeded 120s and whose c-sat execution time didn't exceed 1200s, the geometric mean of parallel speedup was >31 (SAT), >17 (UNSAT) and >22 (overall).

It is quite difficult to make quantitative comparison between different parallel solvers designed to run on different platforms (due to architectural differences, performance variances, etc.), but our experiments gave us the following observations.

SAT-Race 2008 reports the performance of three parallel solvers on four cores [10]. To compare the reported performance with ours, we first measured the performance ratio between the two platforms using the 2007 version of MiniSat. Taking the ratio (the SAT-Race machine being 1.8 times faster rather consistently) into account, it is safe to say that c-sat (i.e., with no tuning of the sequential core but with 31 PEs) with sequential preprocessing solves more problems consistently than ManySat, the winner of the SAT-Race, and will be 2-3 times faster than ManySat. Of the five benchmark suites used in SAT-Race 2008 (excluding the Mixed Suite), c-sat performed best at the Mironov-Zhang

Table 7. Number of problems solved within 1200s by Cluster B and the total runtime

	MiniSat	13PEs	25PEs	37PEs	49PEs	61PEs
# solved (SAT)	46	75	75	79	79	80
# solved (UNSAT)	60	84	89	91	91	95
# solved	106	159	164	170	170	175
total runtime (SAT)	>198573s	24905s	19009s	15995s	15235s	11028s
total runtime (UNSAT)	>181108s	29846s	20641s	20318s	17320s	12535s
total	>379682s	54751s	39650s	36313s	32555s	23563s

Suite in comparison with ManySat (6.5× faster), and then at the Post Suite (3.1× faster). It outperformed ManySat to the least degree (2.1× faster) for the Manolios Suite. Interestingly, the performance characteristics of c-sat were quite close to those of ManySat and quite different from those of pMiniSat and MiraXT, reflecting the solver design.

Comparison with PMSat [4] turned out to be quite difficult because of many parameters/options available. We ran PMSat with 31 PEs with 8 basic options (four methods of assumption generation, each with or without lemma exchange) on 15 problems whose MiniSat execution time ranged from 180s to 908s (SAT) and 518s (UNSAT). The performance of PMSat was highly setting-dependent as reported in [4]. It outperformed c-sat on some SAT problems under some settings, but it seemed difficult to determine the optimal setting in advance. In contrast, c-sat behaved much more stably, and consistently solved all the problems tested, showing its relative strength in UNSAT problems.

3.4 Scalability

It is important to evaluate parallel software using different platforms to find out any platform-dependent peculiarities.

We used Cluster B (with more processors) to evaluate the scalability of c-sat, using the same configuration (i.e., five workers per master). Although the both clusters connect their nodes using gigabit Ethernet and are equipped with exactly the same version of MPICH, Cluster B showed poorer and less stable performance in internode communication even for micro-benchmark problems. With the same configuration as used in Section 3.2 (31 PEs), the communication overhead on Cluster B was around 40% for the SAT-Race 2006 problems as opposed to 10% on Cluster A; the higher overhead was ascertained also with a wall clock. To beat the overhead, we switched off dynamic partitioning and ran 216 problems from SAT-Race 2006 and SAT 2007 Competitions (we did not—though still plan to—use SAT-Race 2008 due to the limited availability of Cluster B). The results are shown in Table 7 and Fig. 9.

In spite of the higher communication overhead, c-sat showed parallel speedup up to 61 PEs. Table 8 shows speedup for problems solved by c-sat within 1200 seconds. The numbers are not directly comparable with the numbers obtained from Cluster A (Fig. 5) due to different time limits on both MiniSat and c-sat

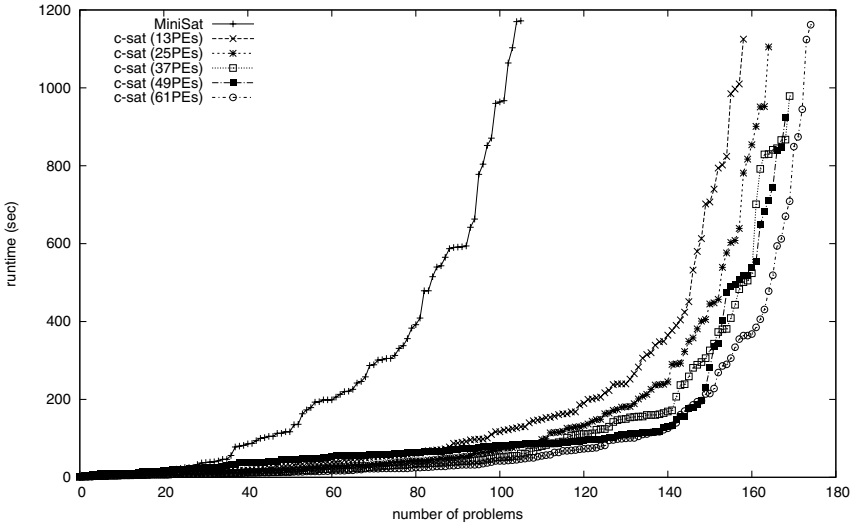


Fig. 9. Execution time with different degree of parallelism (Cluster B)

Table 8. Parallel speedup with Cluster B

	MiniSat	13PEs	19PEs	25PEs	31PEs	37PEs	43PEs	49PEs	55PEs	61PEs
SAT	1	16	21	24	22	30	30	23	36	33
UNSAT	1	7	9	10	12	15	15	15	19	19
total	1	11	14	15	16	21	21	18	26	25

and the different sets of benchmark problems. However, by recalculating and summarizing the numbers of Fig. 5 by using the same time limits, we made sure that Cluster B with 61 PEs showed better parallel speedup than Cluster A with 31 PEs. Finally, the average overhead of lemma exchange (in total runtime) was 39%, but it was only 10% for problems that took more than 120 seconds.

4 Conclusion

We have designed and implemented a cluster-oriented parallel SAT solver c-sat based on MiniSat, and evaluated its performance using a large number of problems. Through several design decisions (on the overall architecture, search and lemma exchange) and tuning, which were both based on a number of preliminary experiments, we obtained at least 31-fold speedup (geometric mean) for satisfiable problems and at least 19-fold speedup for unsatisfiable problems using 31 PEs located on 8 compute nodes connected by gigabit Ethernet. Thus the cluster computing of SAT can be quite efficient (in terms of parallel speedup) and be used to address hard problems without being limited by the current multicore technology. The advantage of c-sat is readily applicable to enhancing the performance of existing bounded model checkers and SAT-based automated planners.

Better stability of performance is another advantage of large-scale solvers. Message passing is considered less efficient than shared-memory, but c-sat realized effective parallel processing with less than 10% of communication overhead for non-small problems.

Our parallelization have deliberately inherited the basic algorithm, heuristics and various parameters of sequential MiniSat and have attained speedup by the tuning of the parallelization part. The same technique should be applicable to other sequential, cutting-edge SAT solvers.

Many things remain to be done in the context of parallel processing, which include: (i) combination with finer-grain parallelism and coarser-grain parallelism (such as algorithm portfolio [7]), (ii) automatic and adaptive tuning of parameters, (iii) using parallelism to ensure the quality of lemmas, and (iv) application to variations of SAT.

References

1. Blochinger, W., Sinz, C., Küchlin, W.: Parallel Propositional Satisfiability Checking with Distributed Dynamic Learning. Parallel Computing 29, 969–994 (2003)
2. Blochinger, W.: Towards Robustness in Parallel SAT Solving. In: Proc. ParCo 2005, John von Neumann Institute for Computing, pp. 301–308 (2006)
3. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
4. Gil, L., Flores, P., Silveira, L.M.: PMSat: a Parallel Version of MiniSAT. JSAT 6, 71–98 (2008)
5. Hamadi, Y., Jabbour, S., Sais, L.: ManySat: Solver Description. Technical Report MSR-TR-2008-83, Microsoft Research (2008)
6. Hyvärinen, A., Junttila, T., Niemelä, I.: Incorporating Learning in Grid-Based Randomized SAT Solving. In: Dochev, D., Pistore, M., Traverso, P. (eds.) AIMSA 2008. LNCS, vol. 5253, pp. 247–261. Springer, Heidelberg (2008)
7. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A Competitive and Cooperative Approach to Propositional Satisfiability. Discrete Applied Mathematics 154(16), 2291–2306 (2006)
8. Lewis, M., Schubert, T., Becker, B.: Multithreaded SAT Solving. In: Proc. 12th Asia and South Pacific Design Automation Conference, pp. 926–931 (2007)
9. Plaza, S., Kountainis, I., Andraus, Z., Bertacco, V., Mudge, T.: Advanced and Insights into Parallel SAT Solving. In: Proc. 15th Int. Workshop on Logic & Synthesis (IWLS 2006), pp. 188–194 (2006)
10. SAT-Race 2008 Results (2008), http://baldur.iti.uka.de/sat-race-2008/results.html
11. Singer, D.: Parallel Resolution of the Satisfiability Problem: A Survey. In: Talbi, E.-G. (ed.) Parallel Combinatorial Optimization, ch. 5. Wiley, Chichester (2006)
12. Zhang, H.: On Subsumption Removal and On-the-Fly CNF Simplification. In: Bacchus, F., Walsh, T. (eds.) SAT 2005, vol. 3569, pp. 482–489. Springer, Heidelberg (2005)
13. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems. J. Symb. Comput. 21(4), 543–560 (1996)
14. Zhang, L., Malik, S.: The Quest for Efficient Boolean Satisfiability Solvers. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 17–36. Springer, Heidelberg (2002)

Author Index