Fahiem Bacchus
Toby Walsh (Eds.)

LNCS 3569

Theory and Applications of Satisfiability Testing

8th International Conference, SAT 2005
St Andrews, UK, June 2005
Proceedings

Springer

Lecture Notes in Computer Science 3569

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

Fahiem Bacchus   Toby Walsh (Eds.)

Theory and Applications of Satisfiability Testing

8th International Conference, SAT 2005
St Andrews, UK, June 19-23, 2005
Proceedings

Springer

Volume Editors

Fahiem Bacchus
University of Toronto, Department of Computer Science
6 King's College Road, Toronto, Ontario, M5S 3H5, Canada
E-mail: fbacchus@cs.toronto.edu

Toby Walsh
National ICT Australia and University of New South Wales
School of Computer Science and Engineering
Sydney 2502, Australia
E-mail: tw@cse.unsw.edu.au

Library of Congress Control Number: 2005927321

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

ISSN 0302-9743
ISBN-10 3-540-26276-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-26276-3 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springeronline.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11499107 06/3142 5 4 3 2 1 0

Preface

The 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005) provided an international forum for the most recent research on the satisfiablity problem (SAT).

SAT is the classic problem of determining whether or not a propositional formula has a satisfying truth assignment. It was the first problem shown by Cook to be NP-complete. Despite its seemingly specialized nature, satisfiability testing has proved to extremely useful in a wide range of different disciplines, both from a practical as well as from a theoretical point of view. For example, work on SAT continues to provide insight into various fundamental problems in computation, and SAT solving technology has advanced to the point where it has become the most effective way of solving a number of practical problems.

The SAT series of conferences are multidisciplinary conferences intended to bring together researchers from various disciplines who are interested in SAT. Topics of interest include, but are not limited to: proof systems and proof complexity; search algorithms and heuristics; analysis of algorithms; theories beyond the propositional; hard instances and random formulae; problem encodings; industrial applications; solvers and other tools.

This volume contains the papers accepted for presentation at SAT 2005. The conference attracted a record number of 73 submissions. Of these, 26 papers were accepted for presentation in the technical programme. In addition, 16 papers were accepted as shorter papers and were presented as posters during the technical programme. The accepted papers and poster papers cover the full range of topics listed in the call for papers.

We would like to thank a number of people and organizations: Ian Miguel, the Local Chair who helped us organize the conference remotely; our generous sponsors who helped us to keep costs down, especially for students; Daniel Le Berre and Laurent Simon for once again organizing the SAT Solver Competition; and Massimo Narizzano and Armando Tacchella for the QBF Solver Evaluation. We would also like to thank the members of the Programme Committee and the additional referees who contributed in the paper-reviewing process.

St Andrews
June 2005

Fahiem Bacchus, Toby Walsh

Organization

Conference Organization

Conference Chairs: Fahiem Bacchus (University of Toronto, Canada)
Toby Walsh (National ICT Australia
and University of NSW, Australia)
Local Chair: Ian Miguel (University of St Andrews, UK)

Programme Committee

Dimitris Achlioptas
Fadi Aloul
Clark Barrett
Constantinos Bartzis
Paul Beame
Armin Biere
Ronen Brafman
Alessandro Cimatti
Adnan Darwiche
Alvaro del Val
Enrico Giunchiglia
Eugene Goldberg
Ziyad Hanna
Edward Hirsch
Henry Kautz
Eleftherios Kirousis
Hans Kleine Büning
Daniel Le Berre
Chu-Min Li
Fangzhen Lin
Sharad Malik
João Marques-Silva
Ilkka Niemela
Toniann Pitassi
Steve Prestwich
Jussi Rintanen
Lakhdar Sais
Karem Sakallah
Laurent Simon
Stefan Szeider
Mirek Truszczynski
Allen Van Gelder
Hans van Maaren
Lintao Zhang

Sponsors

Cadence Design Systems
Intel Corporation
Intelligence Information Systems Institute, Cornell
Microsoft Research
CoLogNet Network of Excellence

Additional Referees

Zaher S. Andraus
Pierre Bonami
Uwe Bubeck
Yin Chen
Sylvie Coste-Marquis
Nadia Creignou
Stefan Dantchev
Sylvain Darras
Gilles Dequen
Laure Devendeville
Niklas Een
Malay K. Ganai

Aarti Gupta
Jean-Luc Guerin
Keijo Heljanko
Jinbo Huang
Dmitry Itsyson
Matti Järvisalo
Tommi Junttila
Bernard Jurkowiak
Zurab Khasidashvili
Arist Kojevnikov
Ioannis Koutis
Alexander Kulikov
Oliver Kullman
Theodor Lettmann
Lengning Liu
Ines Lynce
Yogesh Mahajan
Marco Maratea
Victor Marek
Bertrand Mazure
Maher N. Mneimneh
Alexander Nadel
Massimo Narizzano
Sergey Nikolenko
Nishant Ninha
Thomas Schiex
Armando Tacchella
Muralidhar Talupur
Daijue Tang
Yinlei Yu

Table of Contents

Preface

Solving Over-Constrained Problems with SAT Technology*

Josep Argelich[1] and Felip Manyà[2]

[1] Computer Science Department, Universitat de Lleida,
Jaume II, 69, E-25001 Lleida, Spain
josep@eup.udl.es

[2] Artificial Intelligence Research Institute (IIIA-CSIC),
Campus UAB, 08193 Bellaterra, Spain
felip@iiia.csic.es

Abstract. We present a new generic problem solving approach for over-constrained problems based on Max-SAT. We first define a clausal form formalism that deals with blocks of clauses instead of individual clauses, and that allows one to declare each block either as *hard* (i.e., must be satisfied by any solution) or *soft* (i.e., can be violated by some solution). We then present two Max-SAT solvers that find a truth assignment that satisfies all the hard blocks of clauses and the maximum number of soft blocks of clauses. Our solvers are branch and bound algorithms equipped with original lazy data structures; the first one incorporates static variable selection heuristics while the second one incorporates dynamic variable selection heuristics. Finally, we present an experimental investigation to assess the performance of our approach on a representative sample of instances (random 2-SAT, Max-CSP, and graph coloring).

1 Introduction

The SAT-based problem solving approach presents some limitations when solving many real-life problems due to the fact that it only provides a solution when the formula that models the problem we are trying to solve is shown to be satisfiable. Nevertheless, in many combinatorial problems, some potential solutions could be acceptable even when they violate some constraints. If these violated constraints are ignored, solutions of bad quality are found, and if they are treated as mandatory, problems become unsolvable. This is our motivation to extend the SAT formalism to solve over-constrained problems. In such problems, the goal is to find the *solution* that *best respects* the constraints of the problem.

In this paper we will consider that all the constraints are *crisp* (i.e., they are either completely satisfied or completely violated), but constraints can be

* Research partially supported by projects TIN2004-07933-C03-03 and TIC2003-00950 funded by the *Ministerio de Educación y Ciencia*. The second author is supported by a grant *Ramón y Cajal*.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 1–15, 2005.

either *hard* (i.e., must be satisfied by any solution) or *soft* (i.e., can be violated by some solution). A solution *best respects* the constraints of the problem if it satisfies all the hard constraints and the maximum number of soft constraints. In the literature of over-constrained problems, *fuzzy* constraints (i.e., intermediate degrees of satisfaction are allowed), as well as other ways of defining that a solution *best respects* the constraints of the problem, are considered. We invite the reader to consult [12] for a recent survey on different CSP approaches to solving over-constrained problems.

Given a combinatorial problem which can be naturally defined by a set of constraints over finite-domain variables, we have that each constraint is often encoded as a set (block) of Boolean clauses in such a way that a constraint is satisfiable if all those clauses are satisfied by some truth assignment and is violated if at least one of those clauses is not satisfied by any truth assignment. Thus, in contrast to the usual approach, the concept of satisfaction in SAT-encoded over-constrained problems refers to blocks of clauses instead of individual clauses. This led in turn to design Max-SAT-like solvers that deal with blocks of clauses instead of individual clauses, and exploit the new structure of the encodings.

In this paper we present a new generic problem solving approach for over-constrained problems based on Max-SAT. We first define a clausal form formalism that deals with blocks of clauses instead of individual clauses, and that allows one to declare each block either as *hard* (i.e., must be satisfied by any solution) or *soft* (i.e., can be violated by some solution). We call *soft CNF formulas* to this new kind of formulas. We then present two Max-SAT solvers that find a truth assignment that satisfies all the hard blocks of clauses and the maximum number of soft blocks of clauses. Our solvers are branch and bound algorithms equipped with original lazy data structures; the first one incorporates static variable selection heuristics while the second one incorporates dynamic variable selection heuristics. Finally, we present an experimental investigation to assess the performance of our approach on a representative sample of instances (random 2-SAT, Max-CSP, and graph coloring).

Problem solving of over-constrained problems with Max-SAT local search algorithms has been investigated before in [8, 4]. In that case, the authors distinguish between hard and soft constraints at the clause level, but they do not incorporate the notion of blocks of hard and soft clauses. The notion of blocks of clauses provides a more natural way of encoding soft constraints. Besides, to the best of our knowledge, the treatment of soft constraints with exact Max-SAT solvers has not been considered before.

The paper is structured as follows. In Section 2 we introduce the formalism of soft CNF formulas. In Section 3 we describe a solver for soft CNF formulas with static variable selection heuristics. In Section 4 we describe a solver for soft CNF formulas with dynamic variable selection heuristics. In Section 5 we report the experimental investigation we performed to assess the performance of our formalism and solvers. Finally, we present some concluding remarks.

2 Soft CNF Formulas

We define the syntax and semantics of soft CNF formulas, which are an extension of Boolean clausal forms that we use to encode over-constrained problems.

Definition 1. *A soft CNF formula is formed by a set of pairs $(clause, label)$, where clause is a Boolean clause and label is either h_i or s_i for some $i \in \mathbb{N}$. A hard block of a soft CNF formula is formed by all the pairs $(clause, label)$ with the same label h_i, and a soft block is formed by all the pairs $(clause, label)$ with the same label s_i.*

All the clauses with the same label h_i (s_i) model the same hard (soft) constraint.

Definition 2. *A truth assignment satisfies a hard block of a soft CNF formula if it satisfies all the clauses of the block. A truth assignment satisfies a soft CNF formula ϕ if it satisfies all the hard blocks of ϕ. We say then that ϕ is satisfiable. A soft CNF formula ϕ is unsatisfiable if there is no truth assignment that satisfies all the the hard blocks of ϕ. A truth assignment satisfies a soft block if it satisfies all the clauses of the block. A truth assignment is a solution to a soft CNF formula ϕ if it satisfies all the hard blocks of ϕ and the maximum number of soft blocks.*

Definition 3. *The Soft-SAT problem is the problem of finding a solution to a Soft CNF formula.*

Example 1. We want to solve the problem of coloring a graph with two colors in such a way that the minimum number of adjacent vertices are colored with the same color. If we consider the graph with vertices $\{v_1, v_2, v_3\}$ and with edges $\{(v_1, v_2), (v_1, v_3), (v_2, v_3)\}$, that problem is encoded as a Soft-SAT instance as follows: (i) the set of propositional variables is $\{v_1^1, v_1^2, v_2^1, v_2^2, v_3^1, v_3^2\}$; the intended meaning of variable v_i^j is that vertex v_i is colored with color j; (ii) there is one hard black formed by the following at-least-one and at-most-one clauses:

$$(v_1^1 \vee v_1^2, h_1), (\neg v_1^1 \vee \neg v_1^2, h_1), (v_2^1 \vee v_2^2, h_1), (\neg v_2^1 \vee \neg v_2^2, h_1), (v_3^1 \vee v_3^2, h_1), (\neg v_3^1 \vee \neg v_3^2, h_1);$$

and (iii) there is a soft block for every edge:

$$\begin{array}{c} (\neg v_1^1 \vee \neg v_2^1, s_1), (\neg v_1^2 \vee \neg v_2^2, s_1), \\ (\neg v_1^1 \vee \neg v_3^1, s_2), (\neg v_1^2 \vee \neg v_3^2, s_2), \\ (\neg v_2^1 \vee \neg v_3^1, s_3), (\neg v_2^2 \vee \neg v_3^2, s_3). \end{array}$$

The use of blocks is relevant for two reasons. On the one hand, it provides to the user information in a more natural way about constraint violations. On the other hand, it allows us to get more propagation at certain nodes (this point is discusses in the next section). Besides, the structure of blocks will be important when we extend our formalism to deal with fuzzy constraints.

3 Soft-SAT-S: A Solver with Static Variable Selection Heuristic

The space of all possible assignments for a soft CNF formula ϕ can be represented as a search tree, where internal nodes represent partial assignments and leaf nodes represent complete assignments. The branch and bound algorithm for solving the Max-SAT problem of soft CNF formulas with static variable selection heuristics that we have designed and implemented, called Soft-SAT-S, explores that search tree in a depth-first manner. At each node, the algorithm backtracks if the current partial assignment violates some clause of the hard blocks, and applies the one-literal rule[1] to the literals that occur in unit clauses of hard blocks.[2] If the current partial assignment does not violate any clause of the hard blocks, the algorithm compares the number of soft blocks unsatisfied by the best complete assignment found so far, called upper bound (ub), with the number of soft blocks unsatisfied by the current partial assignment, called lower bound (lb). Obviously, if $ub \leq lb$, a better assignment cannot be found from this point in search. In that case, the algorithm prunes the subtree below the current node and backtracks to a higher level in the search tree. If $ub > lb$, it extends the current partial assignment by instantiating one more variable, say p, which leads to create two branches from the current branch: the left branch corresponds to instantiate p to false, and the right branch corresponds to instantiate p to true. In that case, the formula associated with the left (right) branch is obtained from the formula of the current node by applying the one-literal rule [11] using the literal $\neg p$ (p). The value that ub takes after exploring the entire search tree is the minimum number of soft blocks that cannot be satisfied by a complete assignment.

In branch and bound Max-SAT algorithms like [2, 17], the lower bound is the sum of the number of unsatisfied clauses by the current partial assignment plus an underestimation of the number of clauses that will become unsatisfied if we extend the current partial assignment into a complete assignment, which is calculated taking into account the inconsistency counts of the variables not yet instantiated. The concept of inconsistency counts cannot be easily extended to soft blocks[3] and our lower bound is not so powerful as the lower bounds of [2, 15, 17, 18]. In Soft-SAT-S, like in [2, 17], the initial lower bound is obtained with a GSAT-like [14] local search algorithm.

In Section 5 we define the notion of inconsistency counts for SAT encoded Max-CSP and graph coloring instances by exploiting the structure hidden in the encoding, and we are able to define a lower bound that incorporates an

[1] Given a literal $\neg p$ (p), the one-literal rule [11] deletes all the clauses containing the literal $\neg p$ (p) and removes all the occurrences of the literal p ($\neg p$).

[2] Observe that this pruning technique cannot be applied to exact Max-SAT solvers that deal with individual clauses; in Max-SAT solvers each clause can be viewed as a soft block.

[3] This is due to the fact that a block is unsatisfied by an interpretation I when I does not satisfy one clause of the block.

underestimation of the number of soft blocks that will become unsatisfied if we extend the current partial assignment into a complete assignment.

When branching is done, algorithms for Max-SAT like [2, 17, 3] apply the one-literal rule (simplifying with the branching literal) instead of applying unit propagation (i.e., the repeated application of the one-literal rule until a saturation state is reached) as in the Davis-Putnam-style [6] solvers for SAT. If unit propagation is applied at each node, the algorithm can return a non-optimal solution. For example, if we apply unit propagation to $\{p, \neg q, \neg p \vee q, \neg p\}$ using the unit clause $\neg p$, we derive one empty clause while if we use the unit clause p, we derive two empty clauses. However, when the difference between the lower bound and the upper bound is one, unit propagation can be safely applied, because otherwise by fixing to false any literal of any unit clause we reach the upper bound. Soft-SAT-S performs unit propagation in that case too. Moreover, as pointed out before, Soft-SAT-S applies the one-literal rule when a clause of a hard block becomes unit. This propagation, which leads to substantial performance improvements, cannot be safely applied in Max-SAT solvers like [2, 17, 3], and is a key feature of our approach.

Our current version of Soft-SAT-S incorporates two static variable selection heuristics:

- **MO:** We instantiate first the variables that appears Most Often (MO). Ties are broken using the lexicographical order.
- **csp:** In SAT encodings that model CSP variables, each CSP variable with a domain of size k is represented by a set of k Boolean variables $x_1, \ldots, x_k$. We associate a weight to each one of these sets: the sum of the total number of occurrences of each variable of the set. We order the sets according to such weight. Heuristic **csp** instantiates, first and in lexicographical order, the Boolean variables of the set with the highest weight. Then, it instantiates, in lexicographical order, the Boolean variables of the set with the second highest weight, and so on. This heuristic is used, in the experimental investigation, to solve problems with finite-domain variables (Max-CSP and graph coloring). The idea behind this heuristic is to instantiate first the CSP variables that occur most often. This way, we emulate an n-ary CSP branching by means of a binary branching (i.e., we consider all the possible values of the CSP variable under consideration before instantiating another CSP variable). As we will see in the experiments, we get some performance improvements for the fact of dealing with n-ary branchings.

The fact of using static variable selection heuristics allows us to implement extremely efficient data structures for representing and manipulating soft CNF formulas. Our data structures take into account the following fact: we are only interested in knowing when a clause has become unit or empty. Thus, if we have a clause with four variables, we do not perform any operation in that clause until three of the variables appearing in the clause have been instantiated; i.e., we delay the evaluation of a clause with k variables until $k-1$ variables have been instantiated. In our case, as we instantiate the variables using a static order, we do not have to evaluate a clause until the penultimate variable of the clause in the static order has been instantiated.

The data structures are defined as follows: For each clause we have a pointer to the penultimate variable of the clause in the static order, and the clauses of a soft CNF formula are ordered by that pointer. We also have a pointer to the last variable of the clause. When a variable p is fixed to true (false), only the clauses whose penultimate variable in the static order is $\neg p$ (p) are evaluated. This approach has two advantages: the cost of backtracking is constant (we do not have to undo pointers like in adjacency lists) and, at each step, we evaluate a minimum number of clauses.

4 Soft-SAT-D: A Solver with Dynamic Variable Selection Heuristic

The second solver we have designed and implemented is Soft-SAT-D, which is like Soft-SAT-S except for the fact that its variable selection heuristics are dynamic. This fact, in turn, did not allow us to implement the data structures we have described in the previous section. The data structures implemented in Soft-SAT-D are the two-watched literal data structures of Chaff [13]. They are also lazy data structures, but are not so efficient because here we need to maintain the watched literals.

Our current version of Soft-SAT-D incorporates two dynamic variable selection heuristics:

- **MO:** We instantiate first the variables that appears Most Often (MO). Ties are broken using the lexicographical order. Observe that we do not use the variable that appears most often in minimum size clauses (heuristic MOMS) because this is difficult to know with the lazy data structures of Chaff. However, most of the instances we used in the experimental investigation contain a big amount of binary clauses.
- **MO-csp:** This is the dynamic version of heuristic *csp* of Soft-SAT-S. We associate a weight to each set of free Boolean variables that encode a same CSP variable: the sum of the total number of occurrences of each variable of the set that has not been yet instantiated. We select the set with the highest weight and instantiate its variables in lexicographical order. Like in heuristic *csp*, we emulate an n-ary branching.

5 Experimental Investigation

We next report the experimental investigation we conducted to evaluate the performance of our problem solving approach. All the experiments were performed on a 2GHz Pentium IV with 512 Mb of RAM under Linux.

We performed experiments with ssoft-SAT solvers as well as with weighted Max-SAT solvers and a Max-CSP solver [10]. The solvers used are the following ones:

- Soft-SAT-S with heuristic MO and csp.
- Soft-SAT-D with heuristic MO and MO-csp.
- WMax-SAT: It is a weighted Max-SAT solver that we have implemented. WMax-SAT uses the code of Soft-SAT but does not take into account the notion of hard and soft block; conceptually, WMax-SAT is like Soft-SAT but every clause is treated as a different soft block. The lower bound of WMax-SAT is better than the lower bound of Soft-SAT because it is the sum of the number of unsatisfied clauses by the current partial assignment plus an underestimation of the number of clauses that will become unsatisfied if we extend the current partial assignment into a complete assignment, which is calculated taking into account the inconsistency counts of the variables not yet instantiated. WMax-SAT incorporates the following variable selection heuristic: It instantiates the variables taking into account the number of occurrences in decreasing order (MO).
- BF-improved: It is an improved version of Borchers and Furman's algorithm [3] described in [2]. It uses the popular dynamic variable selection heuristics MOMS (most often in minimum size clauses).
- PFC-MPRDAC: This is a highly optimized solver from the Constraint Programming community for solving binary Max-CSP problems [10].

The benchmarks we used in our experiments are:

- Random 2-SAT instances: We have generated random 2-SAT instances to which we have then assigned, randomly and uniformly, a label corresponding to a hard block or to a soft block. The generator has as parameter the number of blocks: one block is declared to be hard and the rest of blocks are declared to be soft.
- Max-CSP instances: We used SAT-encoded random binary CSPs and solved the Max-CSP problem (the problem of finding an assignment to the variables that satisfies as many constraints as possible). We used Max-CSP instances because they have a natural representation using the formalism of soft CNF formulas. The instances were encoded using the support encoding[4] and generated with a generator of uniform random binary CSPs[5] —designed

[4] In the *support encoding* [9, 7], the idea is to encode into clauses the *support* for a value instead of encoding conflicts. The support for a value j of a CSP variable X_i across a constraint is the set of values of the other variable in the constraint which allow $X_i = j$. If $v_1, v_2, \ldots, v_k$ are the supporting values of variable X_l for $X_i = j$, we add the clause $\neg x_{ij} \vee x_{lv_1} \vee x_{lv_2} \vee \cdots \vee x_{lv_k}$ (called *support* clause). There is one support clause for each pair of variables X_i, X_l involved in a constraint, and for each value in the domain of X_i. We need a similar clause in each direction, one for the pair X_i, X_l and one for X_l, X_i. Besides, we need to add the at-least-one and at-most-one clauses for each CSP variable to ensure that each CSP variable takes exactly one value of its domain. All the at-least-one and at-most-one clauses were encoded as a hard block, and each set of clauses that encodes a CSP constraint was encoded as a different soft block.

[5] http://www.lirmm.fr/~bessiere/generator.html

and implemented by Frost, Bessière, Dechter and Regin— that implements the so-called model B [16]: 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.

– Graph coloring instances: we used unsatisfiable graph coloring instances and the problem we solved was to find a coloring that minimizes the number of adjacent vertices with the same color. We used individual instances from the graph coloring symposium celebrated in CP-2002, and randomly generated instances using the generator of Culberson [5]. We use the generator with option IID (independent random edge assignment). The parameters of the generator are: number of vertices (n), optimum number of colors to get a valid coloring (k), and number of colors we use to color the graph (c).

All the benchmarks encoded as Soft CNF formulas were also encoded as Boolean weighted Max-SAT instances in order to compare our solvers with Boolean weighted Max-SAT solvers. The encoding is as follows: A soft block s_i formed by a set of clauses $\{C_1, \ldots, C_m\}$ is replaced with the set of clauses $\{s_i; 1, C_1 \vee \neg s_i; 2, \ldots, C_m \vee \neg s_i; 2\}$, where s_i is a new Boolean variable and 1 and 2 are weights associated with the clauses. Moreover, we associate a weight $w+1$, where w is the sum of weights of the clauses that encode soft blocks, with each clause belonging to a hard block. Any truth assignment where the sum of unsatisfied clauses is less than $w+1$ is a feasible solution. A solution of a soft CNF formula corresponds to a feasible solution of its weighted Max-SAT encoding with the minimum sum of weights of unsatisfied clauses. Actually, with our encoding, the minimum sum of weights of unsatisfied clauses is identical to the minimum number of soft blocks unsatisfied by a truth assignment that satisfies all the hard blocks.

The Max-CSP instances and the graph coloring instances were also encoded as binary CSP using the format used by PFC-MPRDAC [10], which consists of defining a constraint network by means of a list of nogoods. We believe that it is important to compare our approach with the problem solving approach for over-constrained problems developed in the constraint programming community because they have worked for a long time on this topic and, in contrast to the SAT community, it is a very active research subject.

5.1 Experiments with Random 2-SAT Instances

We compared Soft-SAT-S, Soft-SAT-D and WMax-SAT on random 2-SAT instances using heuristic MO.[6] Figure 1 shows the results for instances with 50 variables and with a number of clauses ranging from 200 to 430, and Figure 2 shows the results for instances with a number of variables ranging from 50 to 100

[6] We tried also to solve the instances with BF-improved, but the results were not competitive.

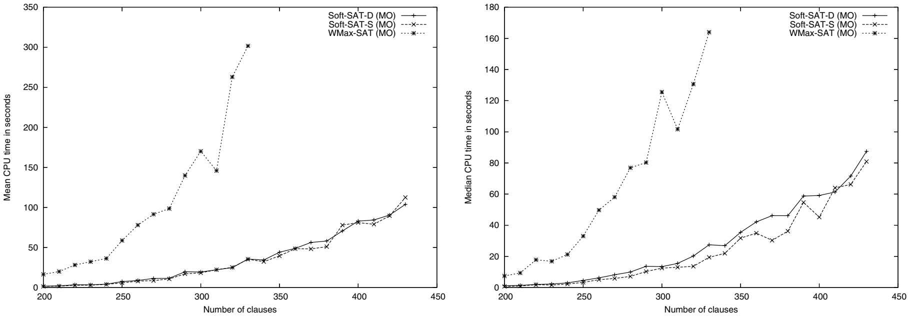


Fig. 1. Random 2-SAT instances with 50 variables and with a number of clauses ranging from 200 to 430

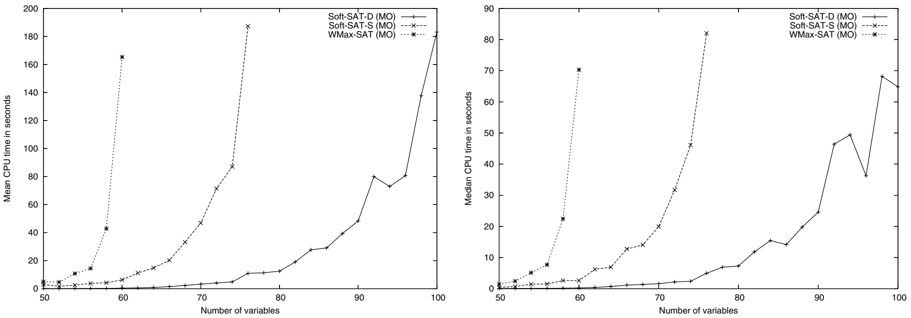


Fig. 2. Random 2-SAT instances with a number of variables ranging from 50 to 100 and with 300 clauses

and with 300 clauses. In both figures we give mean and median time, and each data point corresponds to the time needed to solve a set of 100 instances. In Figure 1 we observe that, in general, Soft-SAT-D is slightly better that Soft-SAT-S, and in Figure 2 we observe that Soft-SAT-D is superior than Soft-SAT-S. WMax-SAT is much worse: Even the fact of having a better lower bound in WMax-SAT does not compensate the extra propagation achieved by exploiting the fact that Soft-SAT knows which clauses encode hard constraints.

5.2 Experiments with Max-CSP Instances

In this section we describe a number of experiments we performed on random binary CSP. For instances with CSP variables with domain size greater than 2, we defined a lower bound that incorporates an underestimation of the number of soft blocks that will become unsatisfied if we extend the current partial assignment into a complete assignment. Each CSP variable with a domain of size k is represented by a set of k Boolean variables $x_1, \ldots, x_k$ in a SAT encoding. The inconsistency count associated with a Boolean variable x_i ($1 \leq i \leq k$) is the

Table 1. Comparison of Soft-SAT-S without underestimation and Soft-SAT-S with underestimation on Max-CSP instances. Time in seconds

	Soft-SAT-S (with underestimation)		Soft-SAT-S (without underestimation)	
$\langle n, d, p_1, p_2\rangle$	mean	median	mean	median
$\langle 10, 15, 45/45, 190/225\rangle$	12.25	10.31	605.03	547.52
$\langle 12, 13, 60/66, 130/169\rangle$	17.94	16.21	2256.91	2010.26
$\langle 13, 8, 78/78, 50/64\rangle$	12.51	11.28	1028.91	973.41
$\langle 15, 10, 50/105, 75/100\rangle$	1.63	1.39	77.54	57.97
$\langle 17, 5, 110/136, 18/25\rangle$	3.35	2.83	394.82	343.61
$\langle 18, 5, 80/153, 18/25\rangle$	0.86	0.76	53.64	46.52
$\langle 20, 5, 90/190, 18/25\rangle$	3.06	2.42	406.53	378.00
$\langle 22, 6, 70/231, 28/36\rangle$	7.10	4.13	910.97	493.15
$\langle 23, 4, 150/253, 12/16\rangle$	16.25	13.59	4615.67	3797.04
$\langle 25, 3, 160/300, 7/9\rangle$	2.66	2.09	142.80	112.51

number of soft blocks violated when x_i is set to true. The inconsistency count associated with a CSP variable X, which is encoded by the Boolean variables $x_1, \ldots, x_k$, is the minimum of the inconsistency counts of x_i $(1 \leq i \leq k)$. As underestimation for the lower bound, we consider exactly one CSP variable for each soft block and take the sum of the inconsistency counts of such variables.

In Table 1 we compare Soft-SAT-S without underestimation with Soft-SAT-S with underestimation for sets of 100 instances of a representative sample of Max-CSP instances. The first column shows the parameters given to the generator of random binary CSPs, and the remaining columns show the experimental results obtained. For each set we give the mean and median time needed to solve an instance of the set. The heuristic used is csp. Table 2 shows the number of backtracks instead of the CPU time for the same instances. In both cases we

Table 2. Comparison of Soft-SAT-S without underestimation and Soft-SAT-S with underestimation on Max-CSP instances. Mean and median number of backtracks

	Soft-SAT-S (with underestimation)		Soft-SAT-S (without underestimation)	
$\langle n, d, p_1, p_2\rangle$	mean	median	mean	median
$\langle 10, 15, 45/45, 190/225\rangle$	2.619.160	2.257.644	807.841.884	735.579.551
$\langle 12, 13, 60/66, 130/169\rangle$	3.432.624	3.005.897	>2.000.000.000	>2.000.000.000
$\langle 13, 8, 78/78, 50/64\rangle$	2.450.851	2.129.608	1.093.257.769	1.168.573.259
$\langle 15, 10, 50/105, 75/100\rangle$	339.848	267.922	141.343.132	96.429.278
$\langle 17, 5, 110/136, 18/25\rangle$	611.488	521.378	564.618.781	520.298.372
$\langle 18, 5, 80/153, 18/25\rangle$	175.118	145.393	114.017.436	92.915.266
$\langle 20, 5, 90/190, 18/25\rangle$	681.346	516.087	601.459.493	631.196.627
$\langle 22, 6, 70/231, 28/36\rangle$	1.750.568	934.992	416.141.039	513.696.823
$\langle 23, 4, 150/253, 12/16\rangle$	2.513.565	2.075.907	>2.000.000.000	>2.000.000.000
$\langle 25, 3, 160/300, 7/9\rangle$	424.359	318.227	337.120.904	262.771.567

Table 3. Comparison of Soft-SAT-S, PFC-MPRDAC and WMax-SAT on Max-CSP instances. Time in seconds

	Soft-SAT-S		PFC-MPRDAC		WMax-SAT	
$\langle n, d, p_1, p_2 \rangle$	mean	median	mean	median	mean	median
$\langle 10, 8, 45/45, 48/64 \rangle$	0.33	0.32	0.19	0.19	11.95	11.41
$\langle 12, 6, 66/66, 27/36 \rangle$	0.48	0.47	0.23	0.23	45.50	45.48
$\langle 14, 5, 91/91, 18/25 \rangle$	0.82	0.77	0.35	0.35	188.75	193.30
$\langle 16, 4, 120/120, 12/16 \rangle$	0.37	0.32	0.22	0.22	275.45	276.25
$\langle 18, 3, 153/153, 6/9 \rangle$	1.00	0.93	0.31	0.31	68.04	62.71
$\langle 15, 6, 60/105, 27/36 \rangle$	0.25	0.24	0.20	0.20	777.94	538.63
$\langle 18, 5, 80/153, 18/25 \rangle$	0.67	0.58	0.33	0.30	5382.68	3364.17
$\langle 20, 5, 70/190, 18/25 \rangle$	0.54	0.44	0.33	0.31	4701.25	2714.86

observe that the fact of adding a lower bound of better quality leads to dramatic performance improvements. In the rest of the paper, all the results reported take into account the above described underestimation.

In the second experiment we compared Soft-SAT-S with heuristic csp,[7] PFC-MPRDAC and WMax-SAT on Max-CSP instances. The results obtained are shown in Table 3. We observe that solver PFC-MPRDAC (which is specialized on solving Max-CSP instances) is about 2 times faster than Soft-SAT-S, but the Weighted Max-SAT approach is much worse. We do not display results with BF-improved because they are worse than the results of WMax-SAT. Even when our solver is not the best, it is quite competitive.

In the third experiment, whose results are shown in Table 4, we solve the same instances of the previous experiment with Soft-SAT-D with heuristic MO-csp and with Soft-SAT-D with heuristic MO in order to compare the n-ary branching with the binary branching. We see that the fact of using an n-ary branching allows us to solve the instances up to 3 times faster. Also observe that Soft-SAT-S (which also uses an n-ary branching) is about 2 times faster than Soft-SAT-D with heuristic MO-csp, and up to 6 times faster than Soft-SAT-D with heuristic MO.

5.3 Experiments with Graph Coloring Instances

The last benchmark we used was graph coloring. In the first experiment we considered 7 sets of randomly generated instances, where each set had 100 instances. We solved the instances with Soft-SAT-S with heuristic csp, Soft-SAT-D with heuristic MO-csp, Soft-SAT-D with heuristic MO, and PFC-MPRDAC.[8] The results obtained are shown in Table 5: the first column displays the parameters

[7] We used this solver of soft CNF formulas because is the best performing one for Max-CSP instances.

[8] We do not give results with Weighted Max-SAT because is not competitive with the solvers used.

Table 4. Comparison of Soft-SAT-D with heuristic MO-csp and Soft-SAT-D with heuristic MO on Max-CSP instances. Time in seconds

	Soft-SAT-D		Soft-SAT-D	
	(MO-csp)		(MO)	
$\langle n, d, p_1, p_2 \rangle$	mean	median	mean	median
$\langle 10, 8, 45/45, 48/64 \rangle$	0.69	0.68	1.83	1.76
$\langle 12, 6, 66/66, 27/36 \rangle$	1.20	1.11	2.92	2.66
$\langle 14, 5, 91/91, 18/25 \rangle$	2.55	2.33	6.82	6.27
$\langle 16, 4, 120/120, 12/16 \rangle$	2.69	2.54	5.44	5.11
$\langle 18, 3, 153/153, 6/9 \rangle$	0.69	0.65	1.40	1.28
$\langle 15, 6, 60/105, 27/36 \rangle$	1.16	1.01	2.21	1.92
$\langle 18, 5, 80/153, 18/25 \rangle$	2.97	2.48	5.98	4.38
$\langle 20, 5, 70/190, 18/25 \rangle$	1.85	1.52	3.80	2.82

Table 5. Comparison between Soft-SAT-S with heuristic csp, Soft-SAT-D with heuristic MO-csp, Soft-SAT-D with heuristic MO, and PFC-MPRDAC on randomly generated graph coloring instances. Time in seconds

	Soft-SAT-S		Soft-SAT-D-MO-csp		Soft-SAT-D-MO		PFC-MPRDAC	
$\langle n, k, c \rangle$	mean	median	mean	median	mean	median	mean	median
$\langle 15, 15, 8 \rangle$	**103.62**	**10.47**	318.75	26.73	743.64	20.12	132.73	21.29
$\langle 15, 15, 10 \rangle$	**102.69**	**0.05**	261.64	0.06	653.74	0.06	139.89	0.15
$\langle 16, 14, 6 \rangle$	**197.13**	**49.00**	986.61	224.60	1350.28	342.58	234.14	78.63
$\langle 16, 14, 8 \rangle$	**164.81**	**19.38**	391.85	29.45	611.52	42.81	207.56	26.26
$\langle 16, 16, 6 \rangle$	**208.48**	**129.61**	950.25	545.06	1503.82	1089.34	250.16	180.93
$\langle 16, 16, 8 \rangle$	**91.87**	**23.33**	224.92	37.11	287.99	46.66	147.50	37.01
$\langle 18, 10, 5 \rangle$	**72.17**	**32.84**	314.09	144.96	435.76	212.44	73.74	42.64

given to the generator, and the rest of columns display the mean and median time needed to solve an instance of the set with each one of the solvers used.

We repeated the previous experiments but using a representative sample of individual instances from the graph coloring symposium celebrated in CP-2002. The results obtained are shown in Table 6: the first column displays the name of the instance, the optimum number of colors to get a valid coloring (k), and the number of colors we used to color the graph (c); the second column displays the number of violated constraints; and the rest of columns display the time needed to solve the instance with each one of the solvers used.

We observe that, in both cases, our approach is superior to the constraint programming approach. For all the sets of randomly generated instances, Soft-SAT-S outperforms PFC-MPRDAC, while for the individual instances some times is better Soft-SAT-S and sometimes Soft-SAT-D. We also observe in both experiments that the n-ary branchings analyzed lead to better performance profiles than the binary branching.

We have also solved some graph coloring instances with the pseudo-Boolean solver PBS v2.1 [1]. Our preliminary results indicate that, at least for the in-

Table 6. Comparison between Soft-SAT-S with heuristic csp, Soft-SAT-D with heuristic MO-csp, Soft-SAT-D with heuristic MO, and PFC-MPRDAC on individual graph coloring instances. Time in seconds

$\langle$Instance, $k, c\rangle$	vc	Soft-SAT-S	Soft-SAT-D-MO-csp	Soft-SAT-D-MO	PFC-MPRDAC
$\langle$myciel5.col, 6, 3$\rangle$	16	**11.04**	46.39	50.47	12.11
$\langle$myciel5.col, 6, 4$\rangle$	4	**78.50**	226.59	344.79	96.41
$\langle$myciel5.col, 6, 5$\rangle$	1	3177.85	**31.87**	73.73	44.34
$\langle$GEOM30a.col, 6, 3$\rangle$	11	**9.31**	27.22	36.11	14.33
$\langle$GEOM30a.col, 6, 4$\rangle$	4	4.48	**2.35**	7.29	22.89
$\langle$GEOM30a.col, 6, 5$\rangle$	1	0.49	**0.15**	0.22	0.18
$\langle$GEOM40.col, 6, 2$\rangle$	22	**3.89**	20.58	21.01	4.42
$\langle$GEOM40.col, 6, 3$\rangle$	7	**10.83**	30.63	65.97	770.46
$\langle$GEOM40.col, 6, 4$\rangle$	3	95.18	**14.67**	69.55	>7200.00
$\langle$GEOM40.col, 6, 5$\rangle$	1	1.58	**0.51**	1.89	1574.44
$\langle$queen5_5.col, 5, 3$\rangle$	29	57.60	167.60	205.37	**27.27**
$\langle$queen5_5.col, 5, 4$\rangle$	12	**37.50**	124.24	148.27	73.67

stances tested, our Soft-SAT solvers outperform PBS. We plan to perform a comprehensive comparison with more pseudo-Boolean solvers in an extended version of this paper.

6 Concluding Remarks

We have presented a new generic problem solving approach for over-constrained problems based on Max-SAT algorithms that deals with hard and soft blocks of clauses. The distinction between hard and soft blocks allows us to model problems in a more natural way, and to traverse the search space of all possible truth assignments in a more efficient way; the extra level of propagation achieved in our solvers is a key factor of the good performance profiles obtained. Our experiments indicate that our approach is better than reducing over-constrained problems to weighted Max-SAT problems. Interestingly, for graph coloring instances, we have shown that the problem solving approach based on Soft CNF formulas also outperforms the approach based on Max-CSP instances. Taking into account the amount of efforts devoted in the constraint programming community on investigating methods of solving over-constrained problems, we believe that the results of this paper open an interesting research avenue.

An important point of our experimental investigation is the good results we obtained due to the lower bound we have implemented, as well as to the fact of using n-ary branching instead of binary branching. It is worth to consider these two points when designing weighted Max-SAT solvers that have to solve more realistic instances. One problem of state-of-the-art Max-SAT solvers is that they are biased to solve randomly generated 2-SAT and 3-SAT instances. They are rarely evaluated with more structured instances and with instances that encode

CSP variables with domain size greater than 2. The results reported here can provide some hints to improve Max-SAT solvers on more realistic instances.

Another important point of our experimental investigation is the good results we obtained with Soft-SAT-S. The extremely efficient data structures that we have implemented are the key of its success. We believe that the incorporation of more sophisticated variable selection heuristics into Soft-SAT-D, will provide us with faster Soft-SAT-D solvers.

As future work, we plan to extend the language of soft CNF formulas to capture fuzzy constraints, to define alternative notions of "the solution that best respects the constraints of the problem", and to incorporate more advanced variable selection heuristics.

It is worth mentioning that we have not found in the SAT literature any approach of solving problems with hard and soft constraints using exact Max-SAT algorithms. All the papers we have found refer to local search algorithms, and do not incorporate the notion of block of clauses.

Finally, we would like to point out that we believe that it is worth exploring how the SAT technology developed for decision problems can be applied to solve optimization problems. This paper has tried to make a step forward in that direction.

References

1. F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo-Boolean solver. In *Symposium on the Theory and Applications of Satisfiability Testing, SAT-2002*, 2002.
2. T. Alsinet, F. Manyà, and J. Planes. Improved branch and bound algorithms for Max-SAT. In *Proceedings of the 6th International Conference on the Theory and Applications of Satisfiability Testing*, 2003.
3. B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2:299–306, 1999.
4. B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki. Local search algorithms for partial MAXSAT. In *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97, Providence/RI, USA*, pages 263–268. AAAI Press, 1997.
5. J. Culberson. Graph coloring page: The flat graph generator. See http://web.cs.ualberta.ca/~joe/Coloring/Generators/flat.html, 1995.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
7. I. P. Gent. Arc consistency in SAT. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI), Lyon, France*, pages 121–125, 2002.
8. Y. Jiang, H. Kautz, and B. Selman. Solving problems with hard and soft constraints using a stochastic algorithm for MAX-SAT. In *Proceedings of the 1st International Workshop on Artificial Intelligence and Operations Research*, 1995.
9. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45:275–286, 1990.
10. J. Larrosa. *Algorithms and Heuristics for Total and Partial Constraint Satisfaction.* PhD thesis, FIB, Universitat Politècnica de Catalunya, Barcelona, 1998.

11. D. W. Loveland. *Automated Theorem Proving. A Logical Basis*, volume 6 of *Fundamental Studies in Computer Science*. North-Holland, 1978.
12. P. Meseguer, N. Bouhmala, T. Bouzoubaa, M. Irgens, and M. Sánchez. Current approaches for solving over-constrained problems. *Constraints*, 8(1):9–39, 2003.
13. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *39th Design Automation Conference*, 2001.
14. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI'92, San Jose/CA, USA*, pages 440–446. AAAI Press, 1992.
15. H. Shen and H. Zhang. Study of lower bound functions for max-2-sat. In *Proceedings of AAAI-2004*, pages 185–190, 2004.
16. B. Smith and M. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155–181, 1996.
17. R. Wallace and E. Freuder. Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems. In D. Johnson and M. Trick, editors, *Cliques, Coloring and Satisfiability*, volume 26, pages 587–615. 1996.
18. Z. Xing and W. Zhang. Efficient strategies for (weighted) maximum satisfiability. In *Proceedings of CP-2004*, pages 690–705, 2004.

A Symbolic Search Based Approach for Quantified Boolean Formulas

Gilles Audemard and Lakhdar Saïs[*]

CRIL CNRS – Université d'Artois,
rue Jean Souvraz SP-18,
F-62307 Lens Cedex France
{audemard, sais}@cril.univ-artois.fr

Abstract. Solving Quantified Boolean Formulas (QBF) has become an important and attractive research area, since several problem classes might be formulated efficiently as QBF instances (e.g. planning, non monotonic reasoning, two-player games, model checking, etc). Many QBF solvers has been proposed, most of them perform decision tree search using the DPLL-like techniques. To set free the variable ordering heuristics that are traditionally constrained by the static order of the QBF quantifiers, a new symbolic search based approach (QBDD(SAT)) is proposed. It makes an original use of binary decision diagram to represent the set of models (or prime implicants) of the boolean formula found using search-based satisfiability solver. Our approach is enhanced with two interesting extensions. First, powerful reduction operators are introduced in order to dynamically reduce the BDD size and to answer the validity of the QBF. Second, useful cuts are achieved on the search tree thanks to the nogoods generated from the BDD representation. Using DPLL-likes (resp. local search) techniques, our approach gives rise to a complete QBDD(DPLL) (resp. incomplete QBDD(LS)) solver. Our preliminary experimental results show that on some classes of instances from the QBF evaluation, QBDD(DPLL) and QBDD(LS) are competitive with state-of-the-art QBF solvers.

Keywords: Quantified boolean formula, Binary decision diagram, Satisfiability.

1 Introduction

Solving quantified boolean formulas has become an attractive and important research area over the last years. Such increasing interest might be related to different reasons including the fact that many important artificial intelligence problems (planning, non monotonic reasoning, formal verification, etc.) can be reduced to QBFs which is considered as the canonical problem of the PSPACE complexity class. Another important reason comes from the recent impressive progress in the practical resolution of the satisfiability problem.

[*] This work has been supported in part by the IUT de Lens, the CNRS and the Region Nord/Pas-de-Calais under the TAC Programme.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 16–30, 2005.

Many solvers for QBFs have been proposed recently (e.g. [11, 18, 12, 10]), most of them are obtained by extending satisfiability results. This is not surprising since QBFs is a natural extension of SAT where the boolean variables are universally or existentially quantified. Most of these solvers take a formula in the prenex clausal form as input and are variant of Davis Logemann and Loveland procedures (DPLL) [7]. However, one of the main drawback of such proposed approaches is that variables are instantiated according to their occurrences in the quantifier prefix (i.e. from the outer to the inner quantifier group). Such preset static ordering limits the efficiency of the search-based QBF solvers. Indeed, in some cases, invalidity of a given QBF might be related to its subparts with variables from the most inner quantifier groups. Consequently, following the order of the prefix might lead to a late and repetitive discovery of the invalidity of the QBF.

The main goal of this paper is to set free the solver from the preset ordering of the QBF and to facilitate the extension of the satisfiability solvers. To this end, binary decision diagrams are used to represent in a compact form the set of models of the boolean formula found by a given satisfiability solver. It give rise to a new QBF solver QBDD(SAT) combining satisfiability search based techniques with binary decision diagram. For completeness and efficiency reasons, our approach is enhanced with two key features. On the one hand, reduction operators are proposed to dynamically reduce at least to some extent the size of the binary decision diagram and to answer the validity of the QBF. On the other hand, for each model found by the satisfiability search procedure, its prime implicant is represented in the BDD and a nogood is returned from the reduced BDD representation and added to the formula. The approach we present in this paper significantly extends our preliminary results [2]. We give a more general framework with additional features such as prime implicants encoding, cuts generation. Two satisfiability search techniques (DPLL and local search techniques) are extended to QBF using our proposed approach.

The paper is organized as follows. After some preliminaries and technical background on quantified boolean formulas and binary decision diagram, it is shown how binary decision diagram can be naturally combined with satisfiability search based techniques to handle QBFs. Using systematic search (respectively stochastic local search) techniques, preliminary experiments on instances of the last QBFs evaluation are presented and show that QBDD(SAT) is competitive and can somtimes achieve significant speedups over state-of-the-art QBF solvers.

2 Preliminaries and Technical Background

Before introducing our approach, we briefly review some necessary definitions and notations about quantified boolean formulas and binary decision diagram.

2.1 Quantified Boolean Formulas

Let $\mathcal{P}$ be a finite set of propositional variables. Then, $\mathcal{L}_{\mathcal{P}}$ is the language of quantified boolean formulas built over $\mathcal{P}$ using ordinary boolean formulas (including propositional constants $\top$ and $\bot$) plus the additional quantification ($\exists$ and $\forall$) over propositional variables.

We consider quantified boolean formula in the prenex form: $\Phi = Q_k X_k, \dots, Q_1 X_1 \Psi$ (in short $QX\Psi$, QX is called the prefix of Φ and Ψ the matrix of Φ) where $Q_i \in \{\exists, \forall\}$, $X_k, \dots, X_1$ are disjoint sets of variables and Ψ a boolean formula. Consecutive variables with the same quantifier are grouped. The rank of a variable $x \in X_i$ is equal to i (noted $rank(x)$). Variables in the same quantifier group have the same rank value. We define a prefix ordering of QBF formula $\Phi = Q_k X_k, \dots, Q_1 X_1 \Psi$ as the partial ordering obtained according to the decreasing rank of the variables, noted $X_k < X_{k-1} < \dots < X_1$. A QBF formula Φ is said to be in clausal form if Φ is in prenex form and Ψ is in Conjunctive Normal Form (CNF). Note that we can consider QBFs with inner quantifier Q_1 as existential. Indeed, if Q_1 is a universal quantifier then suppressing $Q_1 X_1$ from the prefix and all occurrences of $x \in X_1$ from the matrix lead to an equivalent QBF. We define $Var(\Phi) = \bigcup_{i \in \{1,\dots,k\}} X_i$ the set of variables of Φ. A literal is the occurrence of propositional variable in either positive (l) or negative form ($\neg l$). $Lit(\Phi) = \bigcup_{i \in \{1,\dots,k\}} Lit(X_i)$ the set of complete literals of Φ, where $Lit(X_i) = \{x_i, \neg x_i | x_i \in X_i\}$. We note $var(l)$ the variable associated to a literal l. A literal $l \in \Phi$ is a unit literal iff l is existentially quantified and $\exists c = \{l, l_1, \dots, l_i\} \in \Psi$ s.t. $\forall l_j, 1 \leq j \leq i$, $var(l_j)$ is universally quantified and $rank(l_j) < rank(l)$. A monotone literal is defined in the usual way as in the pure boolean case (i.e. l is monotone in Φ iff it appears either positively or negatively).

To define the semantic of quantified boolean formulas, let us introduce some necessary notations. Let S be the set of assignments over the set of variables V. The Up-projection (resp. Down-projection) of a set of assignments S on a set of variables $X \subset V$, denoted $S \uparrow X$ (resp. $S \downarrow X$), is obtained by restricting each assignment to literals in X (resp. in $V \backslash X$). The set of all possible assignments over X is denoted by 2^X. An assignment over X is denoted by a vector of literals $\overrightarrow{x}$. In the same way, Up-projection and Down-projection also apply on vector of literals $\overrightarrow{x}$. If $\overrightarrow{y}$ is an assignment over Y s.t. $Y \cap X = \emptyset$, then $\overrightarrow{y}.S$ denotes the set of interpretations obtained by concatenating $\overrightarrow{y}$ with each interpretation of S. Finally, $\Psi(\overrightarrow{x})$ denotes the boolean formula Ψ simplified with the partial assignment $\overrightarrow{x}$. An assignment $\overrightarrow{x}$ is an implicant or a model (resp. nogood) of Ψ; noted $\overrightarrow{x} \models \Psi$ (resp. $\overrightarrow{x} \not\models \Psi$) iff $\Psi(\overrightarrow{x}) = \top$ (resp. $\Psi(\overrightarrow{x}) = \bot$). An implicant (resp. nogood) $\overrightarrow{x}$ is called prime implicant (resp. minimal nogood) of Ψ iff $\nexists \overrightarrow{y} \subset \overrightarrow{x}$ s.t. $\overrightarrow{y} \models \Psi$ (resp. $\overrightarrow{y} \not\models \Psi$).

A QBF formula is valid (is true) if there exists a solution (called a total policy) defined as follows. It is a simplified version of the definition by Sylvie Coste-Marquis *et al.* [13].

Definition 1. *Let $\Phi = Q_k X_k, \dots, Q_1 X_1 \Psi$ a quantified boolean formula and $\pi = \{\overrightarrow{x}_1, \dots, \overrightarrow{x}_n\}$ a set of models of the boolean formula Ψ. π is a total policy of the quantified boolean formula Φ iff π recursively verifies the following conditions:*

1. *$k = 0$, and $\Psi = \top$*
2. *if $Q_k = \forall$, then $\pi \uparrow X_k = 2^{X_k}$, and $\forall \overrightarrow{x}_k \in 2^{X_k}$, $\pi \downarrow \overrightarrow{x}_k$ is a total policy of $Q_{k-1} X_{k-1}, \dots, Q_1 X_1 \Psi(\overrightarrow{x}_k)$*
3. *if $Q_k = \exists$, then $\pi \uparrow X_k = \{\overrightarrow{x}_k\}$ and $\pi \downarrow \overrightarrow{x}_k$ is a total policy of $Q_{k-1} X_{k-1}, \dots, Q_1 X_1 \Psi(\overrightarrow{x}_k)$*

Remark 1. Let π be a total policy of $\Phi = Q_k X_k, \ldots, Q_1 X_1 \Psi$. If $Q_k = \forall$ then we can rewrite π as $\bigcup_{\overrightarrow{x}_k \in 2^{X_k}} \{\overrightarrow{x}_k.(\pi \downarrow \overrightarrow{x}_k)\}$ and if $Q_k = \exists$, then $\pi \uparrow X_k = \{\overrightarrow{x}_k\}$ and π can be rewritten as $\{\overrightarrow{x}_k.(\pi \downarrow \overrightarrow{x}_k)\}$

Example 1. Let $\Phi = \exists x_5 x_6 \forall x_2 x_4 \exists x_1 x_3 \Psi$ be a QBF formula, where $\Psi = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_4 \vee x_3) \wedge (x_5 \vee x_6)$. Φ is a valid QBF, since the set of models $\pi = \{(\neg x_5, x_6, x_2, x_4, \neg x_1, x_3),(\neg x_5, x_6, x_2, \neg x_4, \neg x_1, x_3), (\neg x_5, x_6, \neg x_2, \neg x_4, x_1, \neg x_3), (\neg x_5, x_6, \neg x_2, x_4, x_1, x_3)\}$ is a total policy of Φ (Figure 1). The different projection operations are illustrated as follow :
$\pi \uparrow \{x_5, x_6\} = \{(\neg x_5, x_6)\}$
$\pi' {=} \pi \downarrow \{x_5, x_6\}$
$= \{(x_2, x_4, \neg x_1, x_3), (x_2, \neg x_4, \neg x_1, x_3), (\neg x_2, \neg x_4, x_1, \neg x_3), (\neg x_2, x_4, x_1, x_3)\}$
$\pi' \uparrow \{x_2, x_4\} {=} \{(\neg x_2, \neg x_4), (\neg x_2, x_4), (x_2, \neg x_4), (x_2, x_4)\} = 2^{\{x_2, x_4\}}$

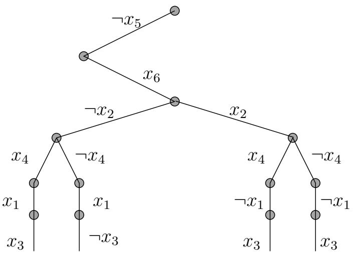


Fig. 1. Policy decision tree representation (example 1)

Motivated by the impressive results obtained in practical solving of the satisfiability problem, several QBF solvers have been developed recently. Most of them are extensions of the well known DPLL procedure including many effective SAT results such as learning, heuristics and constraint propagation (QUBE [11], QUAFFLE [18], EVALUATE [6], DECIDE[16]). For examples, QUBE [11] and QUAFFLE [18] extend backjumping and learning techniques, EVAUATE[6] and DECIDE [16] extend some SAT pruning techniques such as unit propagation.

Algorithm 1 gives a general scheme of a basic DPLL procedure for checking the validity of QBFs. It takes as input variables of the QBF prefix $< X_k, \ldots, X_1 >$ associated to quantifiers $Q_k, \ldots, Q_1$ and a matrix Ψ in clausal form. It returns true if the QBF formula $\Phi = Q_k X_k, \ldots, Q_1 X_1 \Psi$ is valid and false otherwise. The algorithm starts by simplifying the formula using unit propagation and monotone literal rules. Then, if the current simplified matrix contains the empty clause then the current QBF is invalid (value false is returned); otherwise, if the current matrix is empty then the QBF formula is valid (value true is returned). The next step consists in choosing the next variable to instantiate (splitting rule) in the most external non empty set of variables X_k. This differ from the DPLL satisfiability version, since variables are instantiated according to their prefix ordering. Depending on the quantifier Q_k of the chosen variable, left and/or right branchs are generated. If $Q_k = \forall$ (resp. $Q_k = \exists$) the right branch is generated

Algorithm 1: *QDPLL* for QBF

```
Data    : $\Psi$ : matrix of the QBF; $< X_k, \ldots, X_1 >$ : prefix of the QBF $\Phi$
Result  : true if the QBF $\Phi$ s valid, false otherwise
begin
    Simplify($\Psi$);
    if $\emptyset \in \Psi$ then return false;
    if $\Psi = \emptyset$ then return true;
    if $X_k = \emptyset$ then return QDPLL($\Psi, < X_{k-1}, \ldots, X_1 >$);
    choose (by heuristic) a literal $l \in X_k$;
    if (($Q_k = \forall$) and QDPLL($\Psi \cup \{l\}, < X_k - \{l\}, \ldots, X_1 >$)=false) then
        return false;
    if (($Q_k = \exists$) and QDPLL($\Psi \cup \{l\}, < X_k - \{l\}, \ldots, X_1 >$)=true) then
        return true;
    return QDPLL($\Psi \cup \{\neg l\}, < X_k - \{l\}, \ldots, X_1 >$);
end
```

only if the value returned in left branch is true (resp. false). The search tree developped by the QBF DPLL procedure can be seen as an and/or tree search.

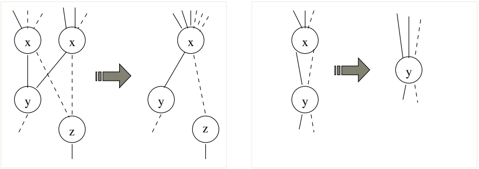
One of the major drawback of the extension of the DPLL procedure to QBF concerns the imposed prefix ordering. Such restrictive ordering might lead to performance degradation of the QBF solver. Since, good ordering might be lost. Furthermore, such limitation makes difficult the extension of certain interesting results obtained on the satisfiability problem (see for example [8] for random problem or [14] for structured problems). One can cite, stochastic local search another search paradigm widely used in SAT (e.g. [17]) that received little attention in QBF. A first integration of local search in QBF solver (WalkQSAT) has been investigated in [10]. WalkQSAT is an implementation of conflict and solution directed backjumping in QBF. It uses a local search solver to guide its search.

2.2 Binary Decision Diagram

A Binary Decision Diagram (BDD) [1, 5] is a rooted directed acyclic graph with two terminal nodes that are referred to as the 0-terminal and the 1-terminal. Every non-terminal node is associated with a primary input variable such that it has two outgoing edges called the 0-edge corresponding to assigning the variable a false truth value, and the 1-edge corresponding to assigning the variable a true truth value. An Ordered Binary Decision Diagram (OBDD) is a BDD such that the input variables appear in a fixed order on all the paths of the graph, and no variable appears more than once in the path. A Reduced Ordered BDD (ROBDD) is an OBDD that results from the repeated application of the following two rules:

1. Share all equivalent sub-graphs (Figure 2.a).
2. Eliminate all redundant nodes whose outgoing edges point to the same node (Figure 2.b).

A (RO)BDD representing a boolean formula Ψ is noted $(RO)BDD(\Psi)$.



a. Sharing sub-formulas b. Redundant node

Fig. 2. Reduction Rules

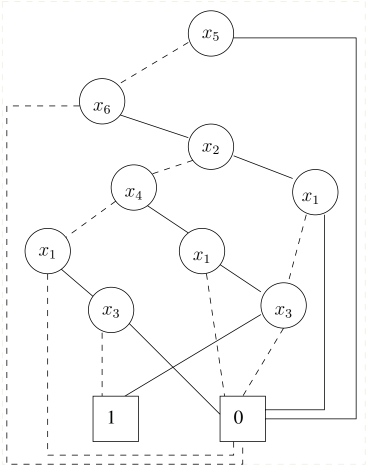


Fig. 3. BDD representation of a policy (example 1)

Figure 3 illustrates the ROBDD representation of the policy π ($ROBDD(\pi)$) shown in example 1 where the solid edges denote the 1-edges and the dashed edges denote the 0-edges. The ROBDD order is the same as the prefix ordering of variables ($\{x_5, x_6\} < \{x_2, x_4\} < \{x_1, x_3\}$) of the QBF Φ.

ROBDDs have some interesting properties. They provide compact and unique representation of boolean functions, and there are efficient algorithms performing all kinds of logical operations on ROBBDs. For example, it is possible to check in constant time whether an ROBDD is true or false. Let us recall that for boolean formula, such problem is NP-complete. Despite the exponential size of the ROBDD in the worst case, ROBDD is one of the most used data structure in practice.

In the rest of this paper, only reduced ordered BDDs are considered and for short we denote them as BDDs. For aquantified boolean formula, the order used in the ROBDD follows the prefix ordering of the QBF.

3 QBDD(SAT): A Symbolic Search Based Approach

In this section, to make the QBF solver freed from the preset ordering of the variables (i.e. fixed by the QBF prefix), we propose an original combination of classical SAT

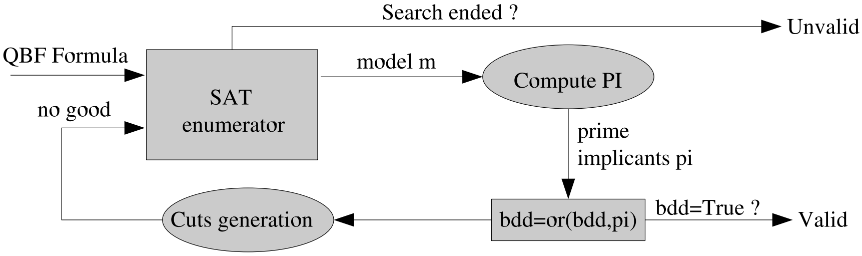


Fig. 4. QBDD(SAT): general scheme

solver with binary decision diagrams. In figure 4, we give a general scheme of our symbolic search based approach QBDD(SAT). More precisely, to check the validity of a QBF $\Phi = QX\Psi$, our approach makes use of a satisfiability technique to search for models (*SAT enumerator*) of the boolean formula Ψ. For each found model m, a prime implicant pi is extracted (*Compute PI*) and disjunctively added to the BDD (*bdd = or(bdd, pi)*) using the prefix ordering of the variables. If the current set of prime implicants represent a total policy then its BDD representation is reduced to 1-terminal node (see section 3.1) and the QBDD(SAT) answer the validity of Φ. As was mentioned earlier, QBDD(SAT) can be instantiated with any satisfiability search technique. For example, QBDD(DPLL) (resp. QBDD(LS)) refer to a QBF solver obtained by instantiating SAT enumerator with DPLL-like (resp. local search) techniques. At the end of the satisfiability search process, if the BDD is not reduced to a 1-terminal node (i.e. a total policy is not found) then depending on the completeness of the SAT used enumerator QBDD(SAT) return either invalid or unknown. Consequently, the QBDD(SAT) is complete iff the satisfiability used solver is also complete.

For space complexity reason, only prime implicants of the boolean formula are encoded in the BDD, nogoods found during the satisfiability search process are not considered. Paths to 0-terminal node in the generated BDD do not represent the nogoods of the boolean formula. Consequently, the 0-terminal node and its incoming edges can be omitted. However, to reduce the search space, for each model (or prime implicant) encoded in the BDD a new nogood is generated and added to the boolean formula (see section 3.2).

3.1 Quantifier Reductions Operators

To reduce the BDD size and to answer the validity of the QBF, additional reduction operator is given in Figure 5. If a node x is existentially quantified and one of its child nodes is the 1-terminal node then any reference to the node x is simply replaced by a reference to its 1-terminal node. We call such reduction operation *existential reduction*. Interestingly enough, when x is universally quantified and its two child nodes are 1-terminal, such node is eliminated using the classical BDD node reduction (Figure 2.b).

During the BDD construction process, in addition to classical reduction operations 2, existential reduction is applied. If the set of models represents a total policy of the QBF formula then the BDD built from such models is reduced to a 1-terminal node as stated by the following property :

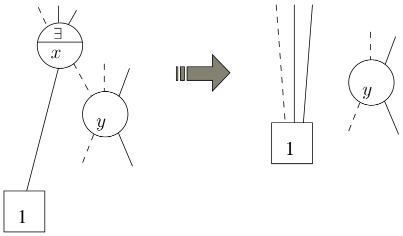


Fig. 5. Existential Reduction

Property 1. Let $\Phi = Q_k X_k, \dots, Q_1 X_1 \Psi$ be a QBF formula and $\pi = \{\overrightarrow{x}_1, \overrightarrow{x}_2, \dots, \overrightarrow{x}_n\}$ a set of models of Ψ. If π is a total policy of Φ then the BDD(π) is reduced to the 1-terminal node.

Proof. The proof is obtained by induction on k. For $k = 0$, the BDD representing the constant $\top$ is a 1-terminal node (by definition of a total policy). Suppose that the property holds for $k - 1$, let us prove that it holds for k. By definition of a total policy two case are considered :

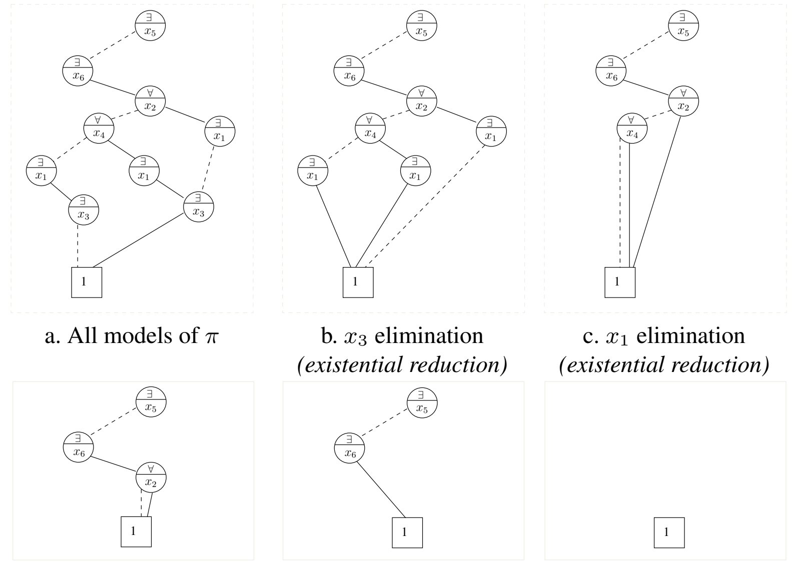
1. if $Q_k = \forall$, then $\pi \uparrow X_k = 2^{X_k}$, and $\forall \overrightarrow{x}_k \in 2^{X_k}$, $\pi \downarrow \overrightarrow{x}_k$ is a total policy of the QBF formula $Q_{k-1}X_{k-1}, \dots, Q_1 X_1 \Psi(\overrightarrow{x}_k)$. By induction hypothesis, we can deduce that BDD($\pi \downarrow \overrightarrow{x}_k$) is reduced to 1-terminal node. Consequently all the leaf of the BDD(2^{X_k}) are 1-terminal nodes. Then by repeatedly applying the Redundant node rule on such a BDD, we obtain a BDD reduced to a 1-terminal node.
2. if $Q_k = \exists$, then $\pi \uparrow \overrightarrow{X}_k = \{\overrightarrow{x}_k\}$ and $\pi \downarrow \overrightarrow{x}_k$ is a total policy of the QBF formula $Q_{k-1}X_{k-1}, \dots, Q_1 X_1 \Psi(\overrightarrow{x}_k)$. By induction hypothesis BDD($\pi \downarrow \overrightarrow{x}_k$) is a 1-terminal node. Consequently, the BDD($\{\overrightarrow{x}_k\}$) can be seen as a branch ended on a 1-terminal node. Repeatedly applying the existential reduction rule, the BDD(π) is reduced to a 1-terminal node.

Remark 2. Dually, *universal reduction* can also be defined. As 0-terminal node of the BDD represents undefined state, then universal reduction operator can not be used in our approach.

The following example shows the dynamic reduction of the BDD associated with the set of models representing the total policy of the QBF given in the example 1.

Example 2. Let Φ be the QBF formula of the example 1 and π its associated policy. The figure 6 represents the reduction phase of the BDD(π) representation:

- The figure 6.a is a BDD representation of the policy π (only paths with final 1-terminal node are represented).
- The existential reduction rule allows the x_3 elimination (figure 6.b) and the x_1 elimination (figure 6.c).
- The redundant node reduction rule allows the x_4 elimination (figure 6.d) and the x_2 elimination (figure 6.e).



a. All models of π

b. x_3 elimination *(existential reduction)*

c. x_1 elimination *(existential reduction)*

d. x_4 elimination *(redundant node reduction)*

e. x_2 elimination *(redundant node reduction)*

f. x_5, x_6 elimination *(existential reduction)*

Fig. 6. BDD reduction of a QBF total policy

- Finally the existential reduction rule suppresses x_5 and x_6 and the bdd of the policy is restricted to the 1-terminal node representing the true formula (figure 6.f) and proving that π is a total policy.

3.2 Generating Cuts from BDD

In order to avoid search of models belonging to different total policies, we introduce in the following different possible cuts (nogoods) that can be generated from the model, prime implicant or from the BDD under construction.

Definition 2. *Let $\Phi = Q_k X_k, \ldots, Q_2 X_2, Q_1 X_1 \Psi$ a QBF s.t. $Q_2 = \forall$, $Q_1 = \exists$ and $\overrightarrow{x}$ is a model of Ψ. We define, $nogood_m(\overrightarrow{x}) = \bigvee\{\neg l | l \in \overrightarrow{x} \downarrow X_1\}$ as the nogood obtained from $\overrightarrow{x}$. In the same way we define $nogood_{pi}(\overrightarrow{pi})$ as the nogood extracted from a prime implicant $\overrightarrow{pi}$.*

Obviously, if $\overrightarrow{pi}$ is a prime implicant obtained from a model $\overrightarrow{x}$ then $nogood_{pi}(\overrightarrow{pi}) \models nogood_m(\overrightarrow{x})$.

Using the example 1, we show in figure 7, that for a model $\overrightarrow{x} = \{\neg x_5, x_6, \neg x_2, x_4, x_1, x_3\}$ the $nogood_m(\overrightarrow{x}) = (x_5 \vee \neg x_6 \vee x_2 \vee \neg x_4)$ ovoid useless search for models of different policies. Considering $\overrightarrow{pi} = \{x_6, x_1, x_3\}$ a prime implicant of $\overrightarrow{x}$, we can generate a strong cut $nogood_{pi}(\overrightarrow{pi}) = \neg x_6$. Interestingly enough, thanks to reductions defined above, the BDD encoding such a prime implicant is reduced to a 1-terminal node and the validity of the QBF is answered.

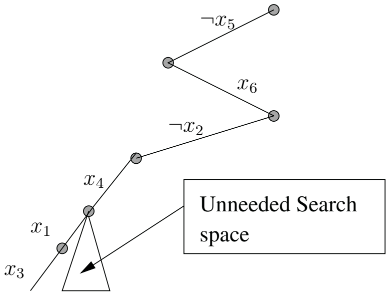


Fig. 7. Cuts generation

Let us recall, that when a prime implicant is disjunctively added to the BDD under construction, the reduction operators eliminate the variables of the inner existential quantifier group X_1. In addition, universally quantified variables in X_2 can be eliminated, when there two outgoing edges point to 1-terminal node. Such a reduction process is iterated recursively. Consequently, to generate strong cuts, in practice each time a prime implicant is added to the BDD, a nogood is extracted from the new reduced BDD and added to the boolean formula.

3.3 QBDD(DPLL) Approach

The algorithm 2 represents the QBF solver obtained by a combination of DPLL procedure with BDD. As we can see, the algorithm search for all models until the BDD representation (characterized by the global variable *bdd* initialized to the 0-terminal node) is reduced to a 1-terminal node, in that case the algorithm terminates and answers the validity of the QBF formula. If the algorithm backtrack to level 0, then the QBF formula is invalid. In other case search continues (back is returned). The function $Simplify()$ enforces the well known unit propagation process. While the function $conflictAnalysis()$ implements learning scheme used by the most efficient satisfiability solvers. Function $primeImplicants()$ extracts a prime implicant from a model I of the boolean formula Ψ. Computing a prime implicant from a given model can be done in linear time. Indeed, for each literal l of the given model m, we verify if the $m\backslash\{l\}$ is also a model of Psi, in such a case the literal l is deleted from m. Finally, $cutsGeneration()$ generates from the current bdd a new nogood and add it to the cnf Ψ (see section 3.2). Example 3 gives a possible trace of QBDD(DPLL) algorithm.

Example 3. Let us consider the formula Φ of example 1. Suppose that the algorithm QBDD(SAT) starts the search by assigning x_1 and x_5. At this step, the clause $(\neg x_4 \vee x_3)$ is satisfied. If we assign a truth value to the literal $\neg x_4$ then a first model $\{x_5, \neg x_4, x_1\}$ is found and is added to the *bdd*. This variable is reduced to a single path to the 1-terminal node : $< x_5, \neg x_4 >$ (variable x_1 is deleted by existential reduction). The clause $(\neg x_5 \vee x_4)$ is added to the formula Ψ. A backtrack is done to search for other models and the assignment of the literal x_4 implies x_3 using unit propagation. A second model $\{x_5, x_4, x_1, x_3\}$ is added to the bdd which becomes reduced to the 1-terminal node using existential and redundant reduction operators. So, search ends and returns the validity of the formula Φ.

Algorithm 2: Combining BDD and DPLL : QBDD(DPLL)

```
Data   : $\Psi$ : set of clauses; X=$\{X_k, \ldots, X_1\}$ the prefix set of the QBF;
         $I$ a partial interpretation ; $d$ level in the search tree, initially set to 0
Result : $valid$ if the QBF is valid, $invalid$ otherwise;
         $back$ is returned to continue the search for other models.
begin
    Simplify($\Psi$);
    if $\emptyset \in \Psi$ then
        conflictAnalysis();
        return $back$;
    if $\Psi = \emptyset$ then
        $pi := primeImplicants(I, \Psi)$;
        $bdd$ := or($bdd$, $pi$);
        if equal(bdd,1-terminal) then return $valid$;
        cutsGeneration($pi$,bdd);
        return $back$;
    Let $l \in X_i$ ($i \in \{1 \ldots k\}$) be the chosen branching variable;
    if (QBDD(DPLL) ($\Psi \cup \{l\}, X - \{l\}, I \cup \{l\}, d + 1$)=valid) or QBDD(DPLL)
    ($\Psi \cup \{\neg l\}, X - \{l\}, I \cup \{\neg l\}, d + 1$)=valid) then
        return $valid$;
    if ($d = 0$) then
        return $invalid$;
    return $back$;
end
```

3.4 QBDD(LS) Approach

One of the important features of our QBDD(SAT) is that any satisfiability search based technique can be used without any major adaptation. Particularly, local search techniques can be integrated in a simple way. Using the state-of-the-art local search WalkSat, we obtain a new incomplete solver QBDD(LS). It answers that the QBF formula is valid, when the set of found models represents a total policy (see property 1); otherwise the solver returns unkown. Our QBDD(LS) solver differs from WalkQSat [10] in the sence that our approach uses local search as a model generator instead of using it to guide the DPLL search procedure.

4 Empirical Evaluation

The experimental results reported in this section are obtained on a Pentium IV 3 GHz with 1GB RAM, and performed on a large panel (644 instances) of the QBF'03 evaluation instances [3]. Theses instances are divided into different families (`log`, `impl`, `toilet`, `k_*`...). For each instance cpu time (in seconds) limit is set to 600 seconds. The QBDD(DPLL) solver is based on chaff like solver called minisat [9], and the QBDD(LS) solver is based on walksat.

4.1 Behaviour of QBDD(DPLL) Solver

Figure 8 presents a pictorial view on the behaviour of different versions of the QBDD (DPLL) solver :

- QBDD(DPLL) is the basic version without computing prime implicants and without generating cuts from the BDD
- QBDD(DPLL) +CUTS is augmented with the generation of cuts from BDD
- QBDD(DPLL) +PI computes prime implicants from a given sat model
- QBDD(DPLL) +PI+CUTS contains these two key features.

The plot in figure 8 is obtained as follows: the x-axis represents the number of benchmarks solved and the y-axis (in log scale) the time needed to solve this number of problems. This figure exhibits clearly that the basic version is the less effective one. Adding only prime implicants does not significantly improve the basic version. However, generating cuts from BDD produces a real improvements, the number of solved problems in less than 600 seconds increases from 130 to 220. Finally, the best version of the QBDD(DPLL) solver is otained by ading cuts and by encoding prime implicants instead of models. Table 1 gives some explanations about these results, it shows the number of models needed to answer the validity of different instances. Since only necessary models are computed with the generation of cuts, the number of models is smaller than without computing cuts. The generation of prime implicants produces some improvements because each prime implicants includes a large potential part of the qbf policies.

Table 2 exhibits the number of instances solved with respect to the size of the quantifier prefix. The method seems to be very efficient with 2QBF formulas and seems to

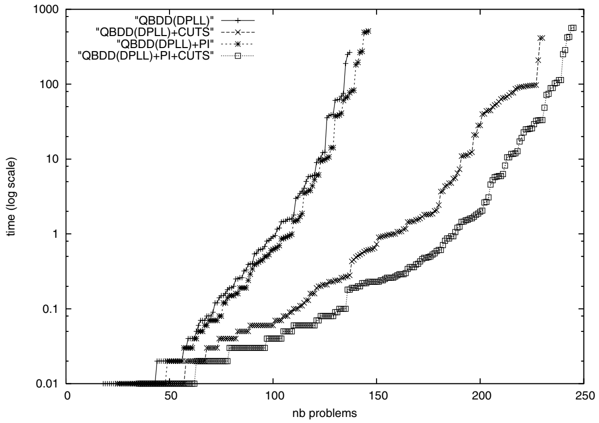


Fig. 8. Number of instances solved vs CPU time

Table 1. Number of models

problem	valid	QBDD(DPLL)	QBDD(DPLL) +PI	QBDD(DPLL) +CUTS	QBDD(DPLL) +PI+CUTS
impl06	Y	6112	6012	2142	550
k_poly_n-1	Y	>45000	>45000	1850	862
toilet_a_10_01.5	N	41412	41412	22486	20 917

Table 2. Solved instances wrt prefix size

prefix size	2	3	4	5	6	7	≥ 8
nb problems	57	339	8	34	7	22	154
nb solved	51	148	1	19	1	11	18
percent	89.5	43.6	12.5	55.8	14.2	50	11.6

be quite ineficcient with large prefix (for example only 11.6% of problems with prefix greater than 8 are solved). However, QBDD(DPLL) is not restricted to 2QBF instances since it solves half of 5-QBF instances. This result agree with those obtained in [15] on 2-QBF using an original combination of two satisfiability solver.

4.2 Comparison with State of the Art Solvers

We compare our solvers QBDD(SAT) to state of the art QBF solvers QUBE [11] and QUANTOR [4]. Here, QBDD(DPLL) solver exploits cuts and prime implicants. Table 3 gives the cpu time comparison between these three solvers on different QBF families instances. *#N* represents the number of problems in the family, *#S* the number of problem solved by a given solver and *TT* the total cpu time needed for a solver to solve all instances of the family. If a method fails to solve an instance then 600 seconds are added to the total cpu time.

Worst results of QBDD(DPLL) are obtained on `toilet` families since only 106 instances are solved in less than 600 seconds, whereas QUBE and QUANTOR solve all quite easily. It's the same for the `k_*` family. Best results are obtained on the `robot` family. Since QBDD(DPLL) solves more and fastly instances than the two other solvers. Furthermore, QBDD(DPLL) solves hard instances of the QBF'03 evaluation for the first time. On a lot of families (`z4`, `flipflop`, `log`) results of all solvers are comparable.

Table 3. CPU time comparison beetwen Qube, Quantor and QBDD(DPLL)

		Qube		Quantor		QBFBDD	
family	#N	#S	TT	#S	TT	#S	TT
robots	48	36	7 214	35	7 970	42	4 270
k_*	171	98	44 813	99	43 786	32	83 658
flipflop	7	7	0.36	7	1.2	7	3.2
toilet	260	260	40	260	256	106	93 860
impl	8	8	0.01	8	0.02	6	1211
tree-exa10	6	6	0.07	6	0.01	6	1.7
chain	8	8	497	8	0.3	2	4183
tree-exa2	6	6	0.01	6	0.01	1	3000.1
carry	2	2	0.23	2	0.69	2	0.71
z4	13	13	0.06	13	0.08	13	0.1
blocks	3	3	0.2	3	0.47	3	3.2
log	2	2	18.4	2	42	2	31

4.3 Comparison with QBDD(LS)

Since QBDD(LS) is an incomplete solver which can not prove the invalidiy of qbf instances, it is quite difficult to make a fair comparison with other qbf solvers. We present its preliminary behaviour on some valid instances and make short comparison with QBDD(DPLL), Quantor and Qube. Table 4 gives the time to solve an instance and the number of models computed during search (if available). For QBDD(LS), each instance is solved 20 times and the median is reported.

Table 4. Comparison on valid instances

	QBDD(LS)		QBDD(DPLL)		Quantor		Qube	
problem	model	time	model	time	model	time	model	time
impl10	3440	8	2673	2	-	0.01	-	0.01
toilet_c_04_10.2	2850	14	?	>600	-	0.01	-	0.01
robots_1_5_2_3.3	58	6	79	0.28	-	453	-	0.7
comp.blif_0.10_1.00_0_1_out_exact	3205	308	4647	1.8	-	0.03	-	>600
tree_exa10-20	?	>600	1095	0.2	-	0.01	-	0.1

These preliminary experiments shows that QBDD(LS) is quite promising. It solves 76 of the 150 valid instances. It is competitive on some QBF instances and obtain better results than other solvers on some other instances.

5 Conclusion

In this paper, a new symbolic search based approach for QBF is presented. It makes an original combination of model search satisfiability techniques with a binary decision diagrams. BDD are used to encode prime implicants of the boolean formula. Reduction operators that prevent to some extent the blowup of the BDD size and answer the validity of the QBF are presented. Interestingly enough, nogoods are generated from the reduced BDD allowing strong cuts in the SAT search space. The main advantage of our approach is that it is freed from any ordering of the variables. This facilitates the extension of satisfiability solver to deal with quantified boolean formulas. Two satisfiability search paradigms (systematic and local search) have been investigated giving rise to two QBF solvers (QBDD(DPLL) and QBDD(LS)). Experimental results on instances from the QBFs evaluation show the effectiveness of our approach. More interestingly, some open hard QBF instances are solved for the first time.

References

1. S. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27:509–516, 1978.
2. Gilles Audemard and Lakhdar Sais. Sat based bdd solver for quantified boolean formulas. In *proceedings of the 16th IEEE international conference on Tools with Artificial Intelligence*, pages 82–89, 2004.

3. Daniel Le Berre, Laurent Simon, and Armando Tachella. Challenges in the qbf arena: the sat'03 evaluation of qbf solvers. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, *LNAI*, pages 452–467, 2003.
4. Armin Biere. Resolve and expand. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2004.
5. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8:677–692, C-35.
6. Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 262–267, Madison (Wisconsin - USA), 1998.
7. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.
8. Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3–sat formulae. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI–01)*, August 4–10 2001.
9. Niklas Eén and Niklas Sörensson. An extensible sat solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, p. 502–508, 2003.
10. Ian P. Gent, Holger H. Hoos, Andrew G. D. Rowley, and Kevin Smyth. Using stochastic local search to solve quantified boolean formulae. In *Proceedings of the 9th international conference of principles and practice of constraint programming*, pages 348–362, 2003.
11. Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QuBE : A system for deciding Quantified Boolean Formulas Satisfiability. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'01)*, Siena, Italy, June 2001.
12. Reinhold Letz. Lemma and model caching in decision procedures for quantified boolean formulas. In *Proceedings of Tableaux 2002*, pages 160–175, Copenhagen, Denmark, 2002.
13. Sylvie Coste Marquis, Helene Fargier, Jerome Lang, Daniel Le Berre, and Pierre Marquis. Function problems for quantified boolean formulas. Technical report, CRIL - France, 2003.
14. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of 38th Design Automation Conference (DAC01)*, 2001.
15. Darsh Ranjan, Daijue Tang and Sharad Malik Niklas. A Comparative Study of 2QBF Algorithms. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2004.
16. Jussi Rintanen. Partial implicit unfolding in the Davis-Putnam procedure for Quantified Boolean Formulae. In *Proceedings of the First International Conference on Quantified Boolean Formulae (QBF'01)*, pages 84–93, 2001.
17. Bart Selman, Hector Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 459–465, 1992.
18. Lintao Zhang and Sharad Malik Towards a symetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 200–215, 2002.

Substitutional Definition of Satisfiability in Classical Propositional Logic

Anton Belov and Zbigniew Stachniak*

Department of Computer Science and Engineering,
York University, Toronto, Canada
{antonb, zbigniew}@cs.yorku.ca

Abstract. The syntactic framework of the so-called saturated substitutions is defined and used to obtain new characterizations of SAT as well as the classes of minimal and maximal models of formulas of classical propositional logic.

1 Introduction

The standard two-valued semantics for classical propositional logic can be 'reconstructed' in the logic's proof theory when the logical constants T and F are chosen to represent the truth-values 1 (*true*) and 0 (*false*). In this approach, substitutions of the logical constants T and F for propositional variables are counterparts of truth-value assignments. Furthermore, a formula $\alpha(p_1, \ldots, p_n)$ is satisfiable if and only if there exists a substitution S of T and F for the variables $p_1, \ldots, p_n$ which, when applied to α, transforms this formula into a tautology. Hence, to establish the satisfiability of α, the search for a satisfying truth-value assignment for α (as in WalkSAT procedure, cf. [11]) can be replaced with the search for a 'satisfying' substitution.

In this paper we consider a construction of 'satisfying' substitutions as an alternative to the search. We begin by providing a construction of a certain type of substitutions which we call *saturated substitutions* for propositional formulas. Saturated substitutions define, in a natural way, truth-value assignments. We show that for every formula α, the class of truth-value assignments defined by all saturated substitutions for α coincides with the class of all satisfying truth-value assignments for α (i.e., of all the models of α). Since the construction of saturated substitutions is provided without any explicit reference to semantics, we obtain a syntactic definition of satisfiability.

The second contribution of this paper is a new characterization of minimal and maximal models of propositional formulas. This characterization is given in terms of the so-called *polarized substitutions*. These saturated substitutions define minimal and maximal models in a very special way. Apart from their definitional completeness (polarized substitutions for a formula α define all and only minimal and maximal models of α), these substitutions carry enough semantic information to flag more truth-value assignments as falsifying a formula α than what explicitly follows from the definitions

* Research supported by the grant from the Natural Sciences and Engineering Research Council of Canada.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 31–45, 2005.

of a minimal or a maximal model of α. In other words, polarized substitutions carry more semantic information about the space of truth-value assignments of propositional formulas than minimal and maximal models. This property of polarized substitutions could be exploited, for instance, in the design of satisfiability-based problem solving methods that require the construction of multiple or all models of propositional formulas (in areas such as model-based diagnosis, model-based reasoning, or planning, cf. [2,5,6,8,10,12], see also [4]).

This paper is structured as follows. In the next section we introduce and discuss the notions of a saturated and polarized substitutions. In Section 3 we provide a syntactic characterization of SAT in the framework of saturated substitutions. Finally, in Section 4, we characterize the classes of minimal and maximal models of formulas of classical propositional logic in terms of polarized saturated substitutions. For reasons of clarity of presentation, the proofs of all the theoretical results stated in this paper are given in the Appendix.

2 Saturated Substitutions

In this section we define the class of saturated substitutions for propositional formulas. Informally speaking, these substitutions are syntactic counterparts of satisfying truth-value assignments – the relationship that we shall explore in the following sections.

We begin with logical preliminaries. The formulas of classical propositional logic are constructed, in the usual way, in terms of propositional variables, logical connectives (negation $\neg$, disjunction $\vee$, conjunction $\wedge$ and, possibly, other connectives), and the logical constants T (*truth*) and F (*falsehood*). By $Var(\alpha)$ we denote the set of all the variables that occur in α. If $Var(\alpha)$ is of cardinality n, then an enumeration of $Var(\alpha)$ is a bijection p from $\{1, \ldots, n\}$ onto $Var(\alpha)$. If p is an enumeration of $Var(\alpha)$, then we shall frequently denote the $i - th$ variable $p(i)$ by p_i. Finally, we shall write $\alpha(p_1, \ldots, p_k)$ to indicate that $p_1, \ldots, p_k$ are some or all the variables from $Var(\alpha)$.

Given a formula α and an enumeration p of its variables, a substitution is a mapping S that assigns a formula $S(p_i)$ to every variable $p_i \in Var(\alpha)$. We shall frequently represent a substitution S as a finite list of the form $[p_1/S(p_1), \ldots, p_n/S(p_n)]$ which explicitly indicates the assignments of formulas $S(p_i)$ to variables p_i, $1 \leq i \leq n$. If $S = [p_1/\alpha_1, \ldots, p_n/\alpha_n]$ is a substitution, then $S(\alpha)$–the application of S to α–is the formula obtained from α by the simultaneous replacement of every occurrence of every variable p_i with α_i. We shall frequently write $\alpha(p_1/\alpha_1, \ldots, p_n/\alpha_n)$ instead of $S(\alpha)$.

A truth-value assignment is a mapping from the set of all propositional variables into $\{0, 1\}$; its extension to all well-formed formulas of propositional logic is defined in the usual way. A truth-value assignment h satisfies a formula α (or h is a *model* of α), if $h(\alpha) = 1$. On the other hand, if $h(\alpha) = 0$, then α is said to be false under h. A formula is satisfiable if it has a model. We denote by SAT the set of all satisfiable formulas. Finally, if h is a truth-value assignment, q is a propositional variable, and $v \in \{0, 1\}$, then by $h[q/v]$ we denote the truth-value assignment defined exactly like h with the exception that $h[q/v](q) = v$.

As mentioned in the introduction, the satisfiability of a formula α is equivalent to the existence of a substitution S which assigns logical constants T and F to the variables of α in such a way that $S(\alpha)$ is a tautology. For every $\alpha \in SAT$ such a satisfying substitution S (called a saturated substitution in the definition given below) can be constructed using the following lemma.

LEMMA 1. *Let α be a propositional formula, let $q \in Var(\alpha)$, and let h be a model of α. Then:*

(i) h is a model of $\alpha(q/\alpha(q/T))$;
(ii) h is a model of $\alpha(q/\neg\alpha(q/F))$.

If h is a model of $\alpha(p_1, \ldots, p_n)$, then, by Lemma 1, it is also a model of $\beta = \alpha(p_1/\gamma, p_2, \ldots, p_n)$, where γ is either $\alpha(p_1/T)$ or $\neg\alpha(p_1/F)$. Note that p_1 does not occur in β. By applying Lemma 1 again, this time to β and p_2, we obtain a satisfiable formula with only $p_3, \ldots, p_n$ as its variables. It should be evident that $n - 2$ additional applications of Lemma 1 to the remaining variables $p_3, \ldots, p_n$ will result in a variable-free formula of the form $\alpha(p_1/\gamma_1, \ldots, p_n/\gamma_n)$ from which a required satisfying substitution S can be extracted by making $S(p_i)$ equivalent to $\gamma_i, 1 \leq i \leq n$. This leads us to the following definition.

DEFINITION 1 (Saturated Substitution). *Let α be a formula and suppose that $Var(\alpha)$ is of cardinality n. Furthermore, let p be an enumeration of $Var(\alpha)$ and let τ be a map from $Var(\alpha)$ into $\{0, 1\}$. A* **substitution string** *for α with respect to p and τ is the sequence of substitutions $S_1, \ldots, S_n$ defined as follows. Let $S_0(\alpha) \overset{def}{=} \alpha$. For every $1 \leq i \leq n$:*

$$S_i(p_i) = \begin{cases} (S_{i-1}(\alpha))(p_i/T), & \text{if } \tau(p_i) = 1, \\ (S_{i-1}(\neg\alpha))(p_i/F), & \text{if } \tau(p_i) = 0. \end{cases}$$

Moreover,

for every $j < i, S_i(p_j) = (S_{i-1}(p_j))(p_i/S_i(p_i))$;
for every $i < j \leq n, S_i(p_j) = p_j$.

The substitution S_n is called a **saturated substitution** *for α.*

The complexity of the definition of a substitution string requires some clarifications. Let us suppose that $S_1, \ldots, S_n$ is the substitution string for $\alpha(p_1, \ldots, p_n)$ with respect to some enumeration p and a mapping τ. First, let us note that for every $1 \leq i \leq n$, the substitution S_i affects only the first i variables $p_1, \ldots p_i$, leaving the remaining variables $p_{i+1}, \ldots p_n$ unchanged. In particular, $S_1(p_1)$ is either $\alpha(p_1/T)$ (when $\tau(p_1) = 1$, see Lemma 1(i)) or $\neg\alpha(p/F)$ (when $\tau(p_1) = 0$, see Lemma 1(ii)). For every $j > 1$, $S_1(p_j) = p_j$. Next, let us observe that for every $i > 1$, we first define $S_i(p_i)$ guided by the mapping τ. Only then we proceed to defining $S_i(p_j)$, for every variable p_j such that $j < i$. Hence, $S_2(p_2)$ is either $(S_1(\alpha))(p_2/T)$ (when $\tau(p_2) = 1$) or $(S_1(\neg\alpha))(p_2/F)$ (when $\tau(p_2) = 0$). Having $S_2(p_2)$ defined, we define $S_2(p_1)$ as $(S_1(p_1))(p_2/S_2(p_2))$, i.e., we substitute $S_2(p_2)$ for p_2 in $S_1(p_1)$. Let us look at the following example.

EXAMPLE 1. Let α be $q \vee r$, let $p_1 = q, p_2 = r$, and let τ be defined by: $\tau(p_1) = 1$ and $\tau(p_2) = 0$. The substitution string for α with respect to p and τ is defined as follows:
$S_1(p_1) = (\alpha(p_1, p_2))(p_1/T) = T \vee p_2$;
$S_1(p_2) = p_2$;
$S_2(p_2) = (\neg S_1(p_1 \vee p_2))(p_2/F) = (\neg(T \vee p_2) \vee p_2))(p_2/F) = \neg((T \vee F) \vee F)$ and is equivalent to F;
$S_2(p_1) = (S_1(p_1))(p_2/S_2(p_2)) = (T \vee p_2)(p_2/S_2(p_2)) = T \vee (\neg((T \vee F) \vee F))$ and is equivalent to T.
S_2 is a saturated substitution for α. ■

Note that if S is a saturated substitution for a formula α, then for every variable $q \in Var(\alpha)$, $S(q)$ is equivalent to either T or F, and, hence, so is $S(\alpha)$.

In Section 4, we shall characterize minimal and maximal models of formulas in terms of polarized substitutions defined as follows. Suppose that S is a saturated substitution for a formula α defined with respect to some enumeration p and a map τ. S is said to be *positive*, if $\tau(q) = 1$, for all $q \in Var(\alpha)$. Similarly, S is said to be *negative*, if $\tau(q) = 0$, for all $q \in Var(\alpha)$. Finally, we shall call S a *polarized substitution* for α, if it is either a positive or a negative saturated substitution for α.

EXAMPLE 2. Consider the formula $\alpha = p_1 \vee p_2$ from Example 1. Let $\tau(p_1) = \tau(p_2) = 1$. The following steps define one of the positive saturated substitutions for α:
$S_1(p_1) = (p_1 \vee p_2)(p_1/T) = T \vee p_2$;
$S_1(p_2) = p_2$;
$S_2(p_2) = (S_1(p_1 \vee p_2))(p_2/T) = ((T \vee p_2) \vee p_2)(p_2/T) = (T \vee T) \vee T$ and is equivalent to T;
$S_2(p_1) = (S_1(p_1))(p_2/S_2(p_2)) = (T \vee p_2)(p_2/S_2(p_2)) = T \vee ((T \vee T) \vee T)$ and is equivalent to T. ■

3 Saturated Substitutions and Propositional Satisfiability

In this section we establish a correspondence between saturated substitutions and satisfying truth-value assignments of propositional formulas (cf. Theorem 1).

DEFINITION 2. *Let α be a formula and let S be one of its saturated substitutions. The truth-value assignment h_S is defined in the following way: for every variable $q \in Var(\alpha)$,*

$$h_S(q) = \begin{cases} 1, & \text{if } q \in Var(\alpha) \text{ and } S(q) \equiv T, \\ 0, & \text{if } q \in Var(\alpha) \text{ and } S(q) \equiv F, \\ \text{arbitrary}, & \text{if } q \notin Var(\alpha). \end{cases}$$

THEOREM 1. *Let α be a propositional formula. Then:*

(i) *if $\alpha \in SAT$, then for every saturated substitution S, h_S is a model of α;*
(ii) *for every model h of α there is a saturated substitution S such that $h = h_S$.*

COROLLARY 2. *Let α be a propositional formula and S be one of its saturated substitutions. Then:*

$$\alpha \in SAT \text{ iff } S(\alpha) \equiv T.$$

In view of Theorem 1, saturated substitutions for a formula α can be considered syntactic counterparts of models of α, that is, models of α can be defined in the syntax of propositional logic in terms of saturated substitutions. Corollary 2 is the characterization of SAT in terms of saturated substitutions: for every saturated substitution S, $S(\alpha)$ is variable free and is equivalent to either T or F; the result of this equivalence test determines the membership in SAT.

4 Minimal and Maximal Models

Minimal and maximal models of propositional formulas are theoretical tools frequently applied in Computer Science in areas such as model-based diagnosis, model-preference default reasoning, or reasoning with models (cf. [5,6,8,10,12]). Maximal and minimal model generation problems have also been widely discussed in the computational complexity literature (cf. [1,7]). In this section we provide a new characterization of these classes of models in terms of polarized substitutions introduced in Section 2. We also prove that polarized substitutions carry, in general, more information about counter-models of propositional formulas than the definitions of minimal and maximal models.

For every propositional formula α we define the relation $\leq_\alpha$ between truth-value assignments in the following way. Given truth-value assignments h_0 and h_1,

$$h_0 \leq_\alpha h_1 \text{ iff for every } p \in Var(\alpha), h_0(p) = 1 \text{ implies } h_1(p) = 1.$$

DEFINITION 3 (Minimal and maximal models). *Let α be a propositional formula. A model h of α is said to be a* **minimal** *(resp. a* **maximal***) model of α, if it is $\leq_\alpha$-minimal (resp. $\leq_\alpha$-maximal).*

THEOREM 3. *Let α be a propositional formula and let h be a truth-value assignment. Then*

(i) *h is a minimal model of α iff $h = h_S$, for some negative saturated substitution S for α;*
(ii) *h is a maximal model of α iff $h = h_S$, for some positive saturated substitution S for α.*

In view of Theorem 3, the definitions of negative (resp. positive) saturated substitutions can be viewed as syntactic definitions of minimal (resp. maximal) models. It turns out that these definitions contain more semantic information about truth-value assignments than merely the fact that they define all the minimal and maximal models of propositional formulas. The following theorem unveils the 'semantic contents' of polarized substitutions.

THEOREM 4 (Blocking Theorem). *Let $\alpha(p_1, \ldots, p_n)$ be a satisfiable formula and let h_{S_+} and h_{S_-} be truth-value assignments defined by a positive and a negative saturated substitutions S_+ and S_-, respectively. Then:*

(i) if for some i, $1 \leq i \leq n$, $h_{S_+}(p_i) = 0$, then for every choice of truth-values $v_1, \ldots, v_{i-1}$,

$$h_{S_+}\big[p_1/v_1, \ldots, p_{i-1}/v_{i-1}, p_i/1\big](\alpha) = 0.$$

(ii) if for some i, $1 \leq i \leq n$, $h_{S_-}(p_i) = 1$, then for every choice of truth-values $v_1, \ldots, v_{i-1}$,

$$h_{S_-}\big[p_1/v_1, \ldots, p_{i-1}/v_{i-1}, p_i/0\big](\alpha) = 0.$$

When searching for multiple models of a formula α, one can use the definition of a particular maximal or a minimal model of α (given in terms of a polarized substitution) to flag some truth-value assignments as falsifying α. In view of the Blocking Theorem, the definitions of polarized substitutions may prune the space of truth-value assignments far more extensively than the definitions of minimal and maximal models alone. We illustrate this comment with the following example.

EXAMPLE 4. Let α be $\neg p_1 \wedge p_2 \wedge p_3$. Consider the negative saturated substitution S (generated with respect to the enumeration p). The minimal model h_S defined by S is given by:

$$h_S(p_1) = 0, h_S(p_2) = 1, h_S(p_3) = 1.$$

Since h_S is a minimal model, the following three truth-value assignments cannot be models of α:

$$\begin{aligned} h_1(p_1) = 0, h_1(p_2) = 0, h_1(p_3) = 0, \\ h_2(p_1) = 0, h_2(p_2) = 0, h_2(p_3) = 1, \\ h_3(p_1) = 0, h_3(p_2) = 1, h_3(p_3) = 0. \end{aligned}$$

However, the Blocking Theorem flags not only these three but also the additional three assignments as falsifying α. These are:

$$\begin{aligned} h_4(p_1) = 1, h_4(p_2) = 0, h_4(p_3) = 0 \text{ (Theorem 4(ii), } i = 3), \\ h_5(p_1) = 1, h_5(p_2) = 1, h_5(p_3) = 0 \text{ (Theorem 4(ii), } i = 3), \\ h_6(p_1) = 1, h_6(p_2) = 0, h_6(p_3) = 1 \text{ (Theorem 4(ii), } i = 2). \end{aligned}$$

There is one more assignment to consider

$$h_7(p_1) = 1, h_7(p_2) = 1, h_7(p_3) = 1.$$

Neither Theorem 4(ii) nor the definition of a minimal model can flag this assignment as falsifying α. However, it is easy to see that h_S can be also defined by any positive substitution S_+ for α (i.e., $h_S = h_{S_+}$). In other words, h_S is also a maximal model of α (cf. Theorem 3). Hence by either maximality of h_S or by Theorem 4(i) ($i = 1$), h_7 falsifies α. To conclude, by applying the Blocking Theorem to any negative substitution S_- and any positive substitution S_+ for α we were able to demonstrate that h_S is the only model of α. ■

To conclude the discussion on Blocking Theorem, let us note that, in general, different enumerations of $Var(\alpha)$ in Definitions 1 and 2 may result in different polarized substitutions for α. Theorem 4, of course, is applicable to all these substitutions.

5 Conclusions and Future Research

This work is a part of a formal theory of logical substitutions which views saturated substitutions as syntactic counterparts of satisfying truth-value assignments. We show that saturated substitutions can be used to obtain new characterizations of SAT as well as the classes of minimal and maximal models of formulas of classical propositional logic.

If α is a satisfiable formula, then its polarized substitutions define all its maximal and minimal models. If α is not satisfiable, then one can still define this formula's saturated substitutions. If, in addition, α is a conjunction of clauses, then, most likely, the polarized substitutions for α define the solutions to MAX-SAT problem of determining the maximal number of clauses of α that can be simultaneously satisfied by some truth-value assignment. The meaning of a polarized substitution for an unsatisfiable free-form formula is unclear at this point but related to the notion of the cumulative clash measure of a formula introduced in [13,14].

Another line of future research concerns the study of saturated substitutions for the purpose of the development of efficient algorithms for SAT, MAX-SAT, and Maximal (Minimal) Model Generation. This includes the complete and incomplete methods as well as algorithms for SAT based on such techniques as the search in Hamming Balls [3] and Satisfiability Coding Lemma [9].

6 Appendix: Technical Results

This Appendix contains the proofs of all the technical results presented in this paper.

Proof of Lemma 1. Let α, q, and h be as stated. The proof of (i) is the case analysis of the value of $h(\alpha(q/T))$.

- If $h(\alpha(q/T)) = 1$, then $h(\alpha(q/\alpha(q/T))) = h(\alpha(q/T)) = 1$.
- If $h(\alpha(q/T)) = 0$, then $h(q) = 0$ (otherwise $h(\alpha(q/T)) = h(\alpha) = 1$). So $h(\alpha(q/T)) = h(q)$ and $h(\alpha(q/\alpha(q/T))) = h(\alpha) = 1$.

The proof of (ii) is similar.

- If $h(\neg\alpha(q/F)) = 0$, then $h(\alpha(q/\neg\alpha(q/F))) = h(\alpha(q/F)) = 1 - h(\neg\alpha(q/F)) = 1$.
- If $h(\neg\alpha(q/F)) = 1$, then $h(q) = 1$ (otherwise $h(\neg\alpha(q/F)) = h(\neg\alpha) = 0$). So $h(\neg\alpha(q/F)) = h(q)$ and $h(\alpha(q/\neg\alpha(q/F))) = h(\alpha) = 1$. ■

LEMMA 2. *Let $S_1, \ldots, S_n$ be the substitution string for a formula α with respect to an enumeration p and a map τ. Then for every $1 \leq i < n$,*

(i) $S_{i+1}(\alpha) = \big[p_{i+1}/S_{i+1}(p_{i+1})\big] S_i(\alpha)$;
(ii) $S_n(p_i) = \big[p_{i+1}/S_n(p_{i+1}), \ldots, p_n/S_n(p_n)\big] S_i(p_i)$.

Proof. Let α and $S_1, \ldots, S_n$ be as stated.
To show (i) let us note that by the definition of a substitution string we have

$$S_{i+1}(\alpha) = \big[p_1/S_{i+1}(p_1), \ldots, p_i/S_{i+1}(p_i), p_{i+1}/S_{i+1}(p_{i+1})\big](\alpha) \qquad (1)$$

Furthermore, for every $j < i + 1$,

$$S_{i+1}(p_j) = \big[p_{i+1}/S_{i+1}(p_{i+1})\big]S_i(p_j).$$

We can therefore rewrite (1) as follows

$$\begin{aligned} S_{i+1}(\alpha) &= \big[p_{i+1}/S_{i+1}(p_{i+1})\big]\big[p_1/S_i(p_1), \ldots, p_i/S_i(p_i)\big](\alpha) \\ &= \big[p_{i+1}/S_{i+1}(p_{i+1})\big]S_i(\alpha). \end{aligned}$$

We prove (ii) in a similar way. By the definition of a substitution string

$$S_n(p_i) = \big[p_n/S_n(p_n)\big]\big[p_{n-1}/S_{n-1}(p_{n-1})\big]\ldots\big[p_{i+1}/S_{i+1}(p_{i+1})\big](S_i(p_i)) \tag{2}$$

When computing the value of $S_n(p_i)$ in (2), each variable p_j, $j > i$, will be replaced by

$$\big[p_n/S_n(p_n)\big]\ldots\big[p_{j+1}/S_{j+1}(p_{j+1})\big]S_j(p_j),$$

which, by the definition of a substitution string, is $S_n(p_j)$. This justifies (ii). ■

LEMMA 3. *Let $S_1, \ldots, S_n$ be the substitution string for a formula α with respect to an enumeration p and a map τ. Furthermore, let $S'_1, \ldots, S'_{n-1}$ be the substitution string for $S_1(\alpha)$ with respect to the enumeration p' of $Var(S_1(\alpha))$ and the map τ' defined as follows:*

- *for every $1 \le i < n, p'_i = p_{i+1}$,*
- *for every $q \in Var(S_1(\alpha))$, $\tau'(q) = \tau(q)$.*

Then, for every $2 \le i \le n$,

(i) $S_i(p_j) = S'_{i-1}(p_j)$*, for every $2 \le j \le i$;*
(ii) $S_i(\alpha) = S'_{i-1}(S_1(\alpha))$.

Proof. We prove (i) and (ii) by induction on i.

Base case: $i = 2$. From the definition of S_2 we have

$$S_2(p_2) = \begin{cases} (S_1(\alpha))(p_2/T), & \text{if } \tau(p_2) = 1, \\ \neg(S_1(\alpha))(p_2/F), & \text{if } \tau(p_2) = 0. \end{cases}$$

Since $p'_1 = p_2$ and $\tau'(p'_1) = \tau(p_2)$, we have

$$S_2(p_2) = \begin{cases} (S_1(\alpha))(p'_1/T), & \text{if } \tau'(p'_1) = 1 \\ \neg(S_1(\alpha))(p'_1/F), & \text{if } \tau'(p'_1) = 0. \end{cases}$$

This means that $S_2(p_2) = S'_1(p'_1) = S'_1(p_2)$, as required.
To demonstrate (ii), let us note that, by Lemma 2(i),

$$S_2(\alpha) = (S_1(\alpha))(p_2/S_2(p_2)).$$

By (i) and the fact that $p_2 = p'_1$, we conclude that

$$S_2(\alpha) = (S_1(\alpha))(p'_1/S'_1(p'_1)) = S'_1(S_1(\alpha)).$$

Inductive step: Assume now that the lemma holds for $i = m < n$. We shall show that the result also holds for $i = m + 1$.

We prove (i) by, first, showing that the result holds for p_{m+1} and, then, for the remaining variables $p_2, \ldots, p_m$.

By the definition of the substitution string $S_1, \ldots, S_n$, we have

$$S_{m+1}(p_{m+1}) = \begin{cases} (S_m(\alpha))(p_{m+1}/T), & \text{if } \tau(p_{m+1}) = 1, \\ \neg(S_m(\alpha))(p_{m+1}/F), & \text{if } \tau(p_{m+1}) = 0. \end{cases}$$

By the inductive hypothesis, $S_m(\alpha) = S'_{m-1}(S_1(\alpha))$. Since $p_{m+1} = p'_m$ and $\tau(p_{m+1})$ $= \tau'(p'_m)$, we get

$$S_{m+1}(p_{m+1}) = \begin{cases} (S'_{m-1}(S_1(\alpha)))(p'_m/T), & \text{if } \tau'(p'_m) = 1, \\ \neg(S'_{m-1}(S_1(\alpha)))(p'_m/F), & \text{if } \tau'(p'_m) = 0, \end{cases}$$

that is,

$$S_{m+1}(p_{m+1}) = S'_m(p'_m) = S'_m(p_{m+1}), \tag{3}$$

as required.

Now, consider a variable p_j, $2 \leq j \leq m$. From the definition of substitution string $S_1, \ldots, S_n$, we have

$$S_{m+1}(p_j) = (S_m(p_j))(p_{m+1}/S_{m+1}(p_{m+1})),$$

which, in view of the inductive hypothesis, the equality (3), and the assumption that $p_{m+1} = p'_m$, gives us

$$S_{m+1}(p_j) = (S'_{m-1}(p_j))(p'_m/S'_m(p'_m)).$$

Since, by the assumption of the lemma, $p_j = p'_{j-1}$, we finally obtain

$$S_{m+1}(p_j) = (S'_{m-1}(p'_{j-1}))(p'_m/S'_m(p'_m)) = S'_m(p'_{j-1}) = S'_m(p_j),$$

as required.

To prove (ii), let us note that, by Lemma 2(i),

$$S_{m+1}(\alpha) = S_m(\alpha)(p_{m+1}/S_{m+1}(p_{m+1})),$$

which, by (3) and the assumption that $p_{m+1} = p'_m$, means that

$$S_{m+1}(\alpha) = S_m(\alpha)(p'_m/S'_m(p'_m)).$$

By the inductive hypothesis, $S_m(\alpha) = S'_{m-1}(S_1(\alpha))$. So

$$S_{m+1}(\alpha) = (S'_{m-1}(S_1(\alpha)))(p'_m/S'_m(p'_m)).$$

By Lemma 2(i), the right hand side of this equation equals $S'_m(S_1(\alpha))$, which gives us the desired $S_{m+1}(\alpha) = S'_m(S_1(\alpha))$. ■

Proof of Theorem 1. Let α be an arbitrary propositional formula and $n = |Var(\alpha)|$. To show (i), let h be a model of α and let $S_1, \ldots, S_n$ be a substitution string for α with respect to some enumeration p of $Var(\alpha)$ and a map τ. We claim that h_{S_n} is a model of α. We first show that for every $1 \leq i \leq n$, h is also a model of $S_i(\alpha)$. We proceed by induction on i.

Base case: $i = 1$. If $\tau(p_1) = 1$, then $S_1(p_1) = \alpha(p_1/T)$. Therefore, $S_1(\alpha) = \alpha(p_1/\alpha(p_1/T))$. Similarly, if $\tau(p_1) = 0$, then $S_1(\alpha) = \alpha(p_1/\neg\alpha(p_1/F))$. Thus,

$$S_1(\alpha) = \begin{cases} \alpha(p_1/\alpha(p_1/T)), & \text{if } \tau(p_1) = 1, \\ \alpha(p_1/\neg\alpha(p_1/F)), & \text{if } \tau(p_1) = 0. \end{cases}$$

Since $h(\alpha) = 1$, by Lemma 1 we must have $h(S_1(\alpha)) = 1$.

Inductive step: Assume now that $h(S_k(\alpha)) = 1$, for some $1 \leq k < n$. Consider the substitution S_{k+1}. By the definition of a substitution string

$$S_{k+1}(p_{k+1}) = \begin{cases} (S_k(\alpha))(p_{k+1}/T), & \text{if } \tau(p_{k+1}) = 1, \\ \neg(S_k(\alpha))(p_{k+1}/F), & \text{if } \tau(p_{k+1}) = 0. \end{cases}$$

By Lemma 2(i), $S_{k+1}(\alpha) = (S_k(\alpha))(p_{k+1}/S_{k+1}(p_{k+1}))$. So,

$$S_{k+1}(\alpha) = \begin{cases} (S_k(\alpha))(p_{k+1}/(S_k(\alpha))(p_{k+1}/T)), & \text{if } \tau(p_{k+1}) = 1, \\ (S_k(\alpha))(p_{k+1}/\neg(S_k(\alpha))(p_{k+1}/F)), & \text{if } \tau(p_{k+1}) = 0. \end{cases}$$

By induction hypothesis, $h(S_k(\alpha)) = 1$. So, using Lemma 1, we get $h(S_{k+1}(\alpha)) = 1$, which completes the inductive proof.

We have just demonstrated that $h(S_n(\alpha)) = 1$. Since $S_n(\alpha)$ is variable-free,

$$T \equiv S_n(\alpha) = [p_1/S_n(p_1), \ldots, p_n/S_n(p_n)](\alpha) \tag{4}$$

Since all the formulas $S_n(p_i)$, $1 \leq i \leq n$, are variable-free (and, hence, equivalent to either T or F), (4) implies that h^* defined as follows: for every $1 \leq i \leq n$,

$$h^*(p_i) = \begin{cases} 1, & \text{if } S_n(p_i) \equiv T, \\ 0, & \text{if } S_n(p_i) \equiv F, \end{cases}$$

is a model of α. But h^* coincides with h_{S_n} on the variables of α. Hence, $h_{S_n}(\alpha) = 1$, which completes the proof of (i).

To show (ii), let h be a model of α, let p be an arbitrary enumeration of $Var(\alpha)$, and let τ be h restricted to $Var(\alpha)$. Let $S_1, \ldots, S_n$ be a substitution string for α with respect to p and τ. We claim that h_{S_n} and h coincide on $Var(\alpha)$, or, equivalently, that for any $1 \leq i \leq n$,

$$S_n(p_i) \equiv \begin{cases} T, & \text{if } h(p_i) = 1, \\ F, & \text{if } h(p_i) = 0. \end{cases} \tag{5}$$

We proceed by induction on n, the number of variables in α.

Base case: Suppose $n = 1$. Let h_0 and h_1 be truth-value assignments such that $h_0(p_1) = 0$ and $h_1(p_1) = 1$.

If $h = h_0$, then $S_1(p_1) = \neg\alpha(p_1/F)$, because $\tau(p_1) = h(p_1) = 0$. Since $h(\alpha) = 1$ and $h(p_1) = 0$ we have $\alpha(p_1/F) \equiv T$ and, hence, $S_1(p_1) = \neg\alpha(p_1/F) \equiv F$, confirming (5). Similarly, if $h = h_1$, then $S_1(p_1) = \alpha(p_1/T)$. Since $h(\alpha) = 1$ and $h(p_1) = 1$, $\alpha(p_1/T) \equiv T$. Thus, $S_1(p_1) = \alpha(p_1/T) \equiv T$, confirming (5).

Inductive step: Assume now that the result holds for any propositional formula of k variables, $k \geq 1$, and that α is a formula of $n = k + 1$ variables. We need to show that for every $1 \leq i \leq k + 1$,

$$S_{k+1}(p_i) \equiv \begin{cases} T, & \text{if } h(p_i) = 1, \\ F, & \text{if } h(p_i) = 0. \end{cases} \tag{6}$$

Consider the substitution S_1. Since $\tau(p_1) = h(p_1)$ and $S_1(p_j) = p_j$, for every $1 < j \leq k + 1$, we have

$$S_1(\alpha) = \begin{cases} \alpha(p_1/\alpha(p_1/T)), & \text{if } h(p_1) = 1, \\ \alpha(p_1/\neg\alpha(p_1/F)), & \text{if } h(p_1) = 0. \end{cases} \tag{7}$$

Since $h(\alpha) = 1$, by Lemma 1 we have $h(S_1(\alpha)) = 1$. Furthermore, $S_1(\alpha)$ is a formula of k variables. Consider the enumeration p' of $Var(S_1(\alpha))$ defined by: for $1 \leq i \leq k$,

$$p'_i = p_{i+1},$$

and the map $\tau' : Var(S_1(\alpha)) \mapsto \{0, 1\}$ defined as follows: for every $1 \leq i \leq k$,

$$\tau'(p'_i) = h(p'_i). \tag{8}$$

By the inductive hypothesis, the substitution S'_k of the string $S'_1, \ldots, S'_k$ for $S_1(\alpha)$ with respect to p' and τ', has the following property: for every $1 \leq i \leq k$,

$$S'_k(p'_i) \equiv \begin{cases} T, & \text{if } h(p'_i) = 1, \\ F, & \text{if } h(p'_i) = 0. \end{cases} \tag{9}$$

It is easy to verify that the substitution strings $S_1, \ldots, S_{k+1}$ and $S'_1, \ldots, S'_k$ satisfy the assumptions of Lemma 3, and, therefore, $S_{k+1}(p_i) = S'_k(p_i)$, for all $2 \leq i \leq k + 1$. Thus, in view of (9), we obtain

$$S_{k+1}(p_i) \equiv \begin{cases} T, & \text{if } h(p_i) = 1, \\ F, & \text{if } h(p_i) = 0, \end{cases} \tag{10}$$

for every $2 \leq i \leq k + 1$.

To complete the the proof it remains to show that

$$S_{k+1}(p_1) \equiv \begin{cases} T, & \text{if } h(p_1) = 1, \\ F, & \text{if } h(p_1) = 0. \end{cases}$$

To this end, let us note that by Lemma 2(ii)

$$S_{k+1}(p_1) = \big[p_2/S_{k+1}(p_2), \dots, p_{k+1}/S_{k+1}(p_{k+1})\big](S_1(p_1)).$$

If $h(p_1) = 1$, then $\tau(p_1) = 1$ and $S_1(p_1) = \alpha(p_1/T)$. Hence

$$\begin{aligned} S_{k+1}(p_1) &= \big[p_2/S_{k+1}(p_2), \dots, p_{k+1}/S_{k+1}(p_{k+1})\big](\alpha(p_1/T)) \\ &= \big[p_1/T, p_2/S_{k+1}(p_2), \dots, p_{k+1}/S_{k+1}(p_{k+1})\big](\alpha) \equiv T. \end{aligned}$$

The last equivalence holds since $h(\alpha) = 1$ and $h(p_1) = 1$. If $h(p_1) = 0$, then the proof is similar. ■

Proof of Theorem 3. Let α be a satisfiable formula of n variables. We shall demonstrate (i) only. The proof of (ii) is similar and is left to the reader.

Let p be some enumeration of $Var(\alpha)$ and let S_+ be a positive saturated substitution for α with respect to p. We must show that h_{S_+} is a maximal model of α.

By Theorem 1, h_{S_+} is a model of α. If $h_{S_+}(q) = 1$, for all $q \in Var(\alpha)$, then h_{S_+} is maximal. Otherwise, the maximality of h_{S_+} follows from Theorem 4(i).

Conversely, suppose that h is a maximal model of α. We shall construct an enumeration p of $Var(\alpha)$ such that for every $1 \leq i \leq n$,

$$S_n(p_i) \equiv \begin{cases} T, & \text{if } h(p_i) = 1, \\ F, & \text{if } h(p_i) = 0, \end{cases} \tag{11}$$

where S_n is the positive saturated substitution for α with respect to p. We proceed by induction on n.

Base case: Suppose that α has just one variable p_1. Let S_1 be the positive saturated substitution for α, that is

$$S_1(p_1) = \alpha(p_1/T). \tag{12}$$

If $h(p_1) = 0$, then $\alpha(p_1/T) \equiv F$, because h is maximal. Hence $S_1(p_1) \equiv F$, as required. If $h(p_1) = 1$, then $\alpha(p_1/T) \equiv T$, because h is a model of α. Hence $S_1(p_1) \equiv T$, as required.

Inductive step: Assume now that (i) holds for any propositional formula of k variables, $k \geq 1$, and that α has $n = k + 1$ variables. We shall construct an enumeration p of $Var(\alpha)$ such that for every $1 \leq i \leq k + 1$,

$$S_{k+1}(p_i) \equiv \begin{cases} T, & \text{if } h(p_i) = 1, \\ F, & \text{if } h(p_i) = 0, \end{cases} \tag{13}$$

where S_{k+1} is the positive saturated substitution for α with respect to p.

Let $q \in Var(\alpha)$ be such that $h(q) = 0$. If there is no such q, then we can repeat the argument presented in the proof of Theorem 1 to show that for every positive saturated substitution S for α, $h = h_S$. Otherwise, consider the formula $\beta = \alpha(q/\alpha(q/T))$. β has k variables and h is also one of its maximal models (this can be demonstrated by induction on the number of variables in β). Thus, by the induction hypothesis, there exists an enumeration p' of $Var(\beta)$, such that for every $1 \leq i \leq k$,

$$S'_k(p'_i) \equiv \begin{cases} T, & \text{if } h(p'_i) = 1, \\ F, & \text{if } h(p'_i) = 0, \end{cases} \tag{14}$$

where S'_k is a positive saturated substitution for β with respect to p'.
Define the enumeration p of $Var(\alpha)$ in the following way:

$$p_i = \begin{cases} q, & \text{if } i = 1, \\ p'_{i-1}, & \text{if } 2 \leq i \leq k+1. \end{cases} \tag{15}$$

Note that the substitution string $S_1, \ldots, S_{k+1}$ for α with respect to p and the substitution string $S'_1, \ldots, S'_k$ for β with respect to p' satisfy the assumptions of Lemma 3. Indeed, since $p_1 = q$, and $\tau(q) = 1$, we have

$$S_1(\alpha) = \alpha(q/\alpha(q/T)) = \beta, \tag{16}$$

and, by (15), $p'_i = p_{i+1}$, for every $1 \leq i \leq k$. The second assumption of Lemma 3 is also satisfied since both strings are positive. So, by Lemma 3(i), for every $2 \leq i \leq k+1$, $S_{k+1}(p_i) = S'_k(p_i)$. Hence, we can use (14) to conclude that (13) holds for every $2 \leq i \leq k+1$.

To complete the proof we only need to show that (13) holds also for $i = 1$ or, equivalently, that $S_{k+1}(p_1) \equiv F$ (since $h(p_1) = h(q) = 0$). To this end, let us note that by Lemma 2(ii),

$$S_{k+1}(p_1) = \left[p_2/S_{k+1}(p_2), \ldots, p_{k+1}/S_{k+1}(p_{k+1})\right](S_1(p_1)).$$

Since $S_1(p_1) = \alpha(p_1/T)$, this means that

$$S_{k+1}(p_1) = \left[p_1/T, p_2/S_{k+1}(p_2), \ldots, p_{k+1}/S_{k+1}(p_{k+1})\right](\alpha).$$

By (13), the maximality of h, and by the assumption that $h(q) = 0$, we conclude that $S_{k+1}(p_1) \equiv h[q/T](\alpha) \equiv F$. ■

Proof of Theorem 4. Let α, h_{S_+}, h_{S_-}, S_+, and S_- be as stated. We shall demonstrate (i) only. The proof of (ii) is similar and is left to the reader. We prove (i) by induction on n, the cardinality of $Var(\alpha)$.

Base case: suppose $n = 1$. Let p_1 be the only variable of α and let S_1 be the positive substitution for α. Since $h_{S_1}(p_1) = 0$, we must have $S_1(p_1) \equiv F$. On the other hand, by the definition of a positive substitution, $S_1(p_1) = \alpha(p_1/T)$. Thus, $\alpha(p_1/T) \equiv F$ which shows that $h_{S_1}\left[p_1/1\right](\alpha) = 0$.

Inductive step: Assume now that (i) holds for any formula of k variables, $k \geq 1$, and that α has $n = k+1$ variables. Let us also assume that $S_+ = S_{k+1}$, for the substitution string $S_1, \ldots, S_{k+1}$ defined for some enumeration p of $Var(\alpha)$. Finally, let us assume that $h_{S_{k+1}}(p_i) = 0$, for some $1 \leq i \leq k+1$. We shall consider the cases $i = 1$ and $1 < i \leq k+1$ separately.

If $i = 1$, then $h_{S_{k+1}}(p_1) = 0$ and, hence, $S_{k+1}(p_1) \equiv F$. On the other hand, by Lemma 2(ii),

$$S_{k+1}(p_1) = \left[p_2/S_{k+1}(p_2), \ldots, p_{k+1}/S_{k+1}(p_{k+1})\right](S_1(p_1)).$$

Since $S_1(p_1) = \alpha(p_1/T)$, we have

$$S_{k+1}(p_1) = \big[p_1/T, p_2/S_{k+1}(p_2), \ldots, p_{k+1}/S_{k+1}(p_{k+1})\big](\alpha).$$

Thus,

$$\big[p_1/T, p_2/S_{k+1}(p_2), \ldots, p_{k+1}/S_{k+1}(p_{k+1})\big](\alpha) \equiv F,$$

and, by the definition of $h_{S_{k+1}}$, we obtain $h_{S_{k+1}}\big[p_1/1\big](\alpha) = 0$, as required.

Now, suppose that $1 < i \leq k+1$. By the definition of a positive saturated substitution, $S_1(p_1) = \alpha(p_1/T)$. Since $S_1(p_j) = p_j$, for the remaining variables of α, we have

$$S_1(\alpha) = \alpha(p_1/S_1(p_1)) = \alpha(p_1/\alpha(p_1/T)). \tag{17}$$

By Lemma 1, $S_1(\alpha)$ is a satisfiable formula of k variables. Let us define the enumeration p' of $Var(S_1(\alpha))$ in the following way: for every $1 \leq j \leq k$,

$$p'_j = p_{j+1}. \tag{18}$$

Consider the substitution string $S'_1, \ldots, S'_k$ for $S_1(\alpha)$ with respect to p' and the map τ' such that for every variable $q, \tau'(q) = 1$. It is straightforward to verify that the substitution strings $S_1, \ldots, S_{k+1}$ and $S'_1, \ldots, S'_k$ satisfy the assumptions of Lemma 3, and, hence, for every $2 \leq j \leq k+1$,

$$S_{k+1}(p_j) = S'_k(p_j). \tag{19}$$

Since, by assumption, $h_{S_{k+1}}(p_i) = 0$, $S_{k+1}(p_i)$ must be equivalent to F. By (19), this means that $S'_k(p_i) \equiv F$ or that $h_{S'_k}(p_i) = 0$. If we put $w = i - 1$, then, in view of (18), we get $h_{S'_k}(p'_w) = 0$.

To summarize, $S_1(\alpha)$ satisfies the assumptions of the inductive hypothesis with respect to $h_{S'_k}$. Hence, for every choice of truth-values $v_1, \ldots, v_{w-1}$,

$$h_{S'_k}\big[p'_1/v_1, \ldots, p'_{w-1}/v_{w-1}, p'_w/1\big](S_1(\alpha)) = 0. \tag{20}$$

By the definition of $h_{S'_k}$, by (18) and (19), for every $1 \leq j \leq k$,

$$h_{S'_k}(p_{j+1}) = \begin{cases} 1, & \text{if } S_{k+1}(p_{j+1}) \equiv T, \\ 0, & \text{if } S_{k+1}(p_{j+1}) \equiv F. \end{cases}$$

In other words, for every $2 \leq j \leq k+1$, $h_{S'_k}(p_i) = h_{S_{k+1}}(p_i)$. From this and the fact that $p_1 \notin Var(S_1(\alpha))$ it follows that (20) can be rewritten as $h_{S_{k+1}}\big[p_2/v_1, \ldots, p_{i-1}/v_{w-1}, p_i/1\big](S_1(\alpha)) = 0$, and, further, using (17), as

$$h_{S_{k+1}}\big[p_2/v_1, \ldots, p_{i-1}/v_{w-1}, p_i/1\big](\alpha(p_1/\alpha(p_1/T))) = 0.$$

By Lemma 1, this implies that neither

$$h_{S_{k+1}}\big[p_2/v_1, \ldots, p_{i-1}/v_{w-1}, p_i/1\big]$$

nor

$$h_{S_{k+1}}\big[p_1/1 - h_{S_{k+1}}(p_1), p_2/v_1, \ldots, p_{i-1}/v_{w-1}, p_i/1\big]$$

are models of α. This completes the proof of (i). ■

References

[1] Z. Chen and S. Toda. The complexity of selecting maximal solutions. *Information and Computation* 119 (1995), pp. 231–239.

[2] H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. *Proc. of AAAI'96* (1996).

[3] E. Dantsin, E.A. Hirsch, and A. Wolper. Algorithms for SAT based on Search in Hamming Balls. *Proc. of STACS 2004, LNCS* 2996 (2004), pp. 141–151.

[4] D.J. Kavvadias and E.C. Stavropoulos. An algorithm for generating all maximal models of Boolean expressions. *Proc. of the 4th Panhellenic Logic Symposium* (2003), pp. 125-129.

[5] R. Khardon and D. Roth. Reasoning with Models. *Artificial Intelligence* 87 (1996), pp. 187-213.

[6] R. Khardon and D. Roth. Default-reasoning with Models. *Proc. of IJCAI'95* (1995), pp. 319–325.

[7] L.M. Kirousis and P.G. Kolaitis. The complexity of minimal satisfiability problems. *Proc. of STACS 2001, LNCS* 2010 (2001), pp. 407–418.

[8] J. de Kleer, A. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence* 56 (1992), pp. 197–222.

[9] R. Paturi, P. Pudlák, and F. Zane. Satisfiability Coding Lemma. *Proc. of FOCS'97* (1997), pp. 566–574.

[10] B. Selman, H.A. Kautz. Model-preference default theories. *Artificial Intelligence* 45 (1990), pp. 287–322.

[11] B. Selman, H. Kautz, and B. Cohen. Noise Strategies for Local Search. *Proc. of AAAI'94* (1994), pp. 337–343.

[12] A. Smith, A. Veneris and A. Viglas. Design Diagnosis Using Boolean Satisfiability. *IEEE Asian-South Pacific Design Automation Conference* (2004).

[13] Z. Stachniak. Exploiting Polarity in Multiple-Valued Inference Systems. *Proc. of the 31st IEEE Int. Symp. on Multiple-Valued Logic* (2001), pp. 149–156.

[14] Z. Stachniak. Going Non-clausal. *Proc. of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing, SAT 2002* (2002), pp. 316–322.

A Clause-Based Heuristic for SAT Solvers

Nachum Dershowitz[1], Ziyad Hanna[2], and Alexander Nadel[1,2]

[1] School of Computer Science,
Tel Aviv University, Ramat Aviv, Israel
{nachumd, ale1}@tau.ac.il
[2] Design Technology Group,
Intel Corporation, Haifa, Israel
{ziyad.hanna, alexander.nadel}@intel.com

Abstract. We propose a new decision heuristic for DPLL-based propositional SAT solvers. Its essence is that both the initial and the conflict clauses are arranged in a list and the next decision variable is chosen from the top-most unsatisfied clause. Various methods of initially organizing the list and moving the clauses within it are studied. Our approach is an extension of one used in Berkmin, and adopted by other modern solvers, according to which only conflict clauses are organized in a list, and a literal-scoring-based secondary heuristic is used when there are no more unsatisfied conflict clauses. Our approach, implemented in the 2004 version of zChaff solver and in a generic Chaff-based SAT solver, results in a significant performance boost on hard industrial benchmarks.

1 Introduction

Propositional satisfiability (SAT) is the problem of determining, for a formula in the propositional calculus, whether there exists a satisfying assignment for its variables. This problem has numerous applications in Formal Verification (e.g., [14]), as well as in Artificial Intelligence (e.g., [8]). SAT solvers are widely used in these and other domains.

Modern complete SAT solvers (e.g., Chaff [10,13], Berkmin [5], Siege [15]) are based on the backtrack-search algorithm of Davis, Putnam, Loveland and Logemann (*DPLL*) [3]. A crucial factor influencing the performance of a DPLL-based SAT solver is its decision heuristic. This heuristic decides which variable to choose at each decision point during the search and what value to assign it first. This paper introduces a new decision heuristic that has been found to be efficient on real-world hard industrial benchmarks. It is designed to increase the likelihood that interrelated variables will be chosen in proximity.

In recent years, the field has been a witness to a breakthrough in the design of decision heuristics that are empirically successful on real-world industrial SAT instances. The key observation was that the decision heuristic must be dynamic, that is, it must re-focus the search on recently derived conflict clauses. VSIDS [13]—the first such dynamic heuristic—maintains a score for each literal. The score is increased when

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 46 – 60, 2005.

the literal appears in a conflict clause; once in a while, scores are halved. This strategy ensures that the prover picks literals that were involved in the derivation of recent conflict clauses.

Another well-known decision heuristic, which proved to be even more successful than VSIDS on benchmark industrial instances, is Berkmin's [5]. Its authors claimed that VSIDS is not sufficiently dynamic, in the sense that it may still pick literals that are irrelevant to the currently explored branch. Instead, they proposed organizing all conflict clauses in a list and picking the next decision literal from the top-most unsatisfied clause on the list. If no such clause exists, a secondary VSIDS-like choice-heuristic is used. Berkmin's heuristic is indeed more dynamic than VSIDS, but we find another advantage of Berkmin's heuristic over VSIDS, in that it tends to pick interrelated variables, that is, variables whose joint assignment increases the chances of both quickly reaching a conflict in an unsatisfiable branch, as well as satisfying and removing "problematic" clauses in satisfiable branches. However, this potential advantage is diluted by the fact that Berkmin does not put the initial clauses on the clause list and applies a secondary VSIDS-like heuristic.

Our proposal, which we call the *clause-based heuristic* (*CBH*), maintains a clause list containing both the initial and the conflict clauses, thus increasing the chances of picking interrelated variables. The next decision literal is picked from the top-most unsatisfied clause. No secondary heuristic is required. We propose various methods of initially organizing the clause list and for moving clauses within it. Our approach results in a significant performance boost over both VSIDS and Berkmin's heuristic.

Basic definitions are provided in the next section. In Section 3, we review the DPLL algorithm [3], enhanced by Boolean constraint propagation [17] and conflict clause recording [12]. Readers familiar with this material—included in the paper to make it self-contained—may skip ahead. Some of the commonly used decision heuristics are summarized in Section 4. In Section 5, we present our clause-based heuristic. In Section 6, we report on experiments that show that CBH is superior to VSIDS, Berkmin's decision heuristic and the Berkmin-like decision heuristic of zChaff-2004, when tested on hard industrial benchmarks used in the SAT'04 competition [9]. This is followed by a brief conclusion.

2 Basic Definitions

In what follows, we denote negation by $\neg$, conjunction by $\wedge$ and disjunction by $\vee$. A *literal* is a variable or its negation. We use p, q, r, etc., for literals. A *clause* is a disjunction of literals, usually denoted by letters A, B, ... The number of literals in a clause A is denoted by $|A|$. A *conjunctive normal form (CNF)* formula is a conjunction of clauses, like $\varphi = (p) \wedge (\neg p \vee q) \wedge (\neg q \vee \neg r)$. It is sometimes convenient to represent a CNF formula as a set of clauses, rather than as a conjunction. For example, an alternative way to represent φ is as the set $\{\, p, \neg p \vee q, \neg q \vee \neg r \,\}$. We denote an empty clause by $\varnothing$.

An (*partial*) *assignment* (σ) is a (partial) function assigning a truth value, 1 or 0, to all (some) variables. An example is the assignment $\tau(p) = \tau(r) = 1$, $\tau(q) = 0$. It is sometimes convenient to consider a (partial) assignment as a subset of literals. For example, an alternative way to represent τ is as $\{p, \neg q, r\}$. A CNF formula φ evaluates under partial assignment σ to $\sigma[\varphi]$ by applying the following rules for each variable p:

1. If $\sigma(p) = 1$, remove $\neg p$ from φ's clauses and remove clauses containing p from φ.
2. If $\sigma(p) = 0$, remove p from φ's clauses and remove clauses containing $\neg p$ from φ.

If $\sigma[\varphi]$ contains no clauses, then $\sigma[\varphi] = 1$ and σ is a *satisfying assignment* or a *model* for φ, and we say that $\varphi=1$ *under* σ. If $\emptyset \in \sigma[\varphi]$, then σ is an *unsatisfying assignment* for φ, and we say that $\varphi=0$ *under* σ. Formula φ is *satisfiable* if there exists an assignment σ, such that $\varphi=1$ under σ; otherwise, it is *unsatisfiable.*

3 Enhanced DPLL

The DPLL algorithm [3] is the basic backtrack-search algorithm for SAT. DPLL is based upon the validity of the following two rules:

1. *Splitting Literal Rule*: Let $\sigma = \{p\}$ and $\tau = \{\neg p\}$. Formula φ is satisfiable iff $\sigma[\varphi]$ or $\tau[\varphi]$ is satisfiable.
2. *Unit Clause Rule:* Let $\sigma = \{p\}$. If there is a clause $A \in \varphi$ such that $A = p$, then φ is satisfiable iff $\sigma[\varphi]$ is.

Boolean constraint propagation (BCP) [17] is a process of extending a partial assignment by repeatedly applying the unit-clause rule. A *decision variable* is a variable assigned as a result of an application of the splitting rule. A *decision literal* is the assigned literal of a decision variable. An *implied variable* is a variable assigned as a result of the BCP process. An *implied literal* is the assigned literal of an implied variable. The *decision level* of an assigned variable p is one more than the number of decision variables assigned prior to p.

Let φ be an input formula. Modern complete SAT solvers implement the DPLL algorithm in the following way: A partial assignment σ, initialized to the empty partial assignment, is maintained. The partial assignment σ is extended by executing the BCP process. Then, if the input formula φ is neither 1 nor 0 under σ, the splitting rule is used, that is, the satisfiability of φ is recursively checked under $\sigma\cup\{p\}$ and then under $\sigma\cup\{\neg p\}$, as required. The literal p is picked using some *decision heuristic.*

Next we describe a powerful technique, used by all the modern SAT solvers, namely *conflict clause recording* [12]. A *conflict* is a situation during DPLL invocation when a formula φ evaluates to 0 under the current assignment σ. After such a conflict, it is possible to identify an assignment $\tau \subseteq \sigma$, such that if χ is an assignment containing τ, extended by invoking BCP on $\tau[\varphi]$, then φ is 0 under χ. In other words, applying τ to φ is sufficient to create the same conflict as applying σ to φ up to BCP. We refer to such a partial assignment τ as a *conflict-sufficient assignment.* Let τ be a conflict-sufficient assignment. Then a clause A consisting of the negations of all the literals contained in τ is referred to as a *conflict clause*. The process of adding conflict clauses to the formula after each conflict is referred to as *conflict clause recording*. Conflict-clause recording prevents DPLL from re-exploring the same sub-space during the subsequent search. Different *conflict recording schemes* add conflict clauses corresponding to different conflict-sufficient assignments. The 1UIP conflict recording scheme [12,13] has empirically been shown to be the most efficient [15,18]. In what follows, we describe the 1UIP conflict clause identification.

First, we define an "implication clause" for an assigned literal p. Let σ be a partial assignment held by DPLL prior to assigning p. The *implication clause* of an implied literal p, denoted by $ic(p)$, is the clause that became a unit under σ and made the implication of p possible, during BCP. The implication clause of a decision literal p is the empty clause $\varnothing$. After a conflict, a *conflicting variable* r is the variable that had been assigned last, before it was identified that σ is an unsatisfying assignment. Both literals of a conflicting variable are referred to as *conflicting literals*. According to the definition of an unsatisfying assignment, an empty clause must belong to $\sigma[\varphi]$. It means that there is a clause $F \in \varphi$ that is 0 under σ. One of the conflicting literals must belong to F, otherwise r would not have been the last variable that was assigned a value. We denote a conflicting literal that belongs to F by $ecl(r)$. For example, if $\varphi = (p \vee r) \wedge (p \vee \neg r)$; $\sigma = \{\neg p, r\}$, then $\sigma[\varphi] = 0$; $p \vee \neg r$ is the 0-clause under σ; r is the conflicting variable and $ecl(r) = \neg r$. Observe that the implication clause is well defined for $\neg ecl(r)$, since $\neg ecl(r)$ is an assigned literal. However, the implication clause is not defined for $ecl(r)$, since $ecl(r)$ is not an assigned literal. We extend the definition of the implication clause by setting $ic(ecl(r)) = F$. Note that by the extended definition, the conflicting variable has two implication clauses (one for each literal.)

An *implication graph* is a directed acyclic graph, in which each vertex is associated with a literal corresponding to an assignment along with its decision level. There is a directed edge from vertex p to vertex q if $ic(q) = A$ and $\neg p \in A$. Consider the example in Fig. 1, which illustrates concepts presented in this section. DPLL has been invoked on formula φ and the situation at the time of the first conflict is shown in Table 1. The implication graph is shown at the bottom. The decision literals $\neg y$, $\neg p$ and $\neg s$ do not have incoming edges. Any implied literal y has $|\, ic(y)\,| - 1$ incoming edges.

Now we are ready to describe the concept of conflict recording schemes. Any conflict clause is generated by a *conflict cut* of the implication graph. All the decision literals are on one side of the conflict cut (called the *reason side*). Both conflicting literals are on the other side of the conflict cut (called the *conflict side*). All vertices on the reason side that have at least one edge to the vertices of the conflict side comprise the conflict's *reason*. Let $\tau \subseteq \sigma$ be a partial assignment containing only the assigned literals corresponding to the reason of a conflict cut. The partial assignment τ is a conflict-sufficient assignment.

Next, we describe the *1UIP learning scheme*. Let p and q be two literals assigned at the same decision level dl. Then, in an implication graph, p is said to *dominate* q, if and only if any path from the decision variable corresponding to level dl to q contains p. A *unique implication point* (*UIP*) [12] is an assigned literal of the highest decision level dominating both literals of the conflicting variable. Observe that the decision literal corresponding to the highest decision level is always a UIP. The 1UIP learning scheme requires that any literal assigned after the first UIP (counting from the conflicting literals) will be on the conflict side of the corresponding conflict cut and all others will be on the reason side. Consider again the example in Fig. 1. The literals t

and $\neg s$ are UIP's, since both dominate x and $\neg x$. The literal t is the first UIP, since it is closer than $\neg s$ to the conflicting literals. The 1UIP cut is shown on Fig. 1. The corresponding conflict clause is $\neg q \vee \neg t$.

$$\begin{gathered}A_1 = \neg w \vee s \vee v \vee \neg q;\ A_2 = p \vee q;\ A_3 = \neg q \vee r \vee s \vee \neg w;\ A_4 = \neg q \vee \neg t \vee x; \\ A_5 = s \vee \neg r \vee \neg v \vee t;\ A_6 = \neg t \vee \neg x;\ A_7 = y \vee w; \\ \varphi = A_1 \wedge A_2 \wedge A_3 \wedge A_4 \wedge A_5 \wedge A_6 \wedge A_7.\end{gathered} \tag{1}$$

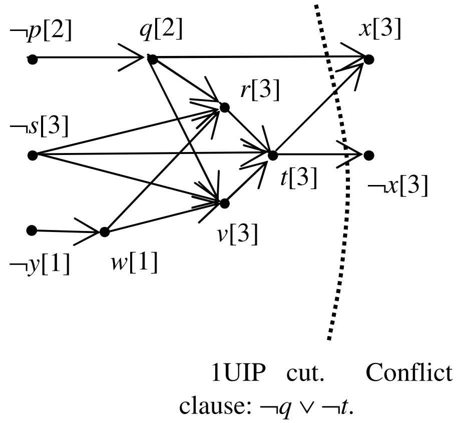


Fig. 1. An example of an implication graph and 1UIP conflict clause identification. The implication graph corresponds to a DPLL invocation on Formula (1). The related variable values, implication clauses and decision level of the variables are shown on Table 1

Table 1. The implication graph and 1UIP conflict clause identification principles. The value, decision level and implication clause for each variable of formula (1) are shown. The corresponding implication graph and 1UIP conflict cut are displayed in Fig. 1

	p	q	r	s	t	v	w	x	y
σ	0	1	1	0	1	1	1	1	0
DL	2	2	3	3	3	3	1	3	1
ic	$\emptyset$	2	3	$\emptyset$	5	1	7	4	$\emptyset$

4 Existing Decision Heuristics

In this section, we describe the most widely used decision heuristics, known to be efficient on real-world industrial benchmarks.

Early *static* heuristics (e.g. Jeroslaw-Wang [7], Literal Count [11]) picked the next variable based on the number of appearances (scores) of different variables in unsatisfied clauses. A major drawback of such an approach is that score calculation requires visiting all the clauses at each node of the search tree, which implies a very significant overhead. Another disadvantage of static heuristics is that they do not consider information that can be retrieved after a conflict during implication graph analysis.

Heuristics based upon such analyses were found to be several orders of magnitude faster [5,13].

The first *dynamic* heuristic is called Variable State Independent Decaying Sum (VSIDS) [13]. According to VSIDS, each literal is associated with a counter *cl*(*p*), whose value is increased once a new clause containing *p* is added to the database. Counters are initialized to 0. Every once in a while, all counters are halved. The next literal to be picked is the one with the largest counter. Ties are broken randomly. Two major advantages of VSIDS over the previous heuristics are that: (1) VSIDS is characterized by a negligible computational cost; (2) VSIDS gives preference to literals that participate in recent conflicts, i.e. it is dynamic. MiniSat SAT solver [4] implements a variant of VSIDS. Instead of infrequent halving of the scores, MiniSat multiplies the scores after each conflict by 0.95. This makes the heuristic more dynamic.

The authors of the Berkmin SAT solver [5] proposed a successful decision heuristic that has been partially or fully adopted by the most modern SAT solvers, such as the 2004 version of zChaff [10], Satzoo [4], and Oepir [1]. We show, in the experimental section, that Berkmin's heuristic is indeed faster than VSIDS on hard industrial benchmarks. The main difference of Berkmin's heuristic, when compared to VSIDS, is as follows: Conflict clauses are organized in a list, and every new conflict clause is appended to the head of the list. The next decision variable is picked from the top-most unsatisfied clause. If no such clause exists, the next decision variable is chosen according to a VSIDS-like heuristic. We will describe Berkmin's heuristic in detail and analyze why it is advantageous over VSIDS.

Berkmin maintains a counter $cl'(p)$ measuring the contribution of each literal to the search. Unlike VSIDS, Berkmin augments $cl'(p)$, not only for literals that belong the conflict clause itself, but also for literals that belong to one of the clauses that were traversed in the implication graph during 1UIP conflict clause identification. At intervals, Berkmin divides all the counters by 4 (compared to 2 for VSIDS). Let $cv'(p)$ be a counter measuring the contribution of each variable to the conflicts, defined as $cl'(p) + cl'(\neg p)$. Berkmin maintains all conflict clauses in a list. After each conflict, the new conflict clause is appended to the top of the list. The next decision variable is one with the highest $cv'(p)$ out of all the variables of the topmost unsatisfied clause. If no conflict clauses exist yet, or if all the conflict clauses are satisfied, then the variable with the highest $cv'(p)$ of all unassigned variables is chosen.

Next, we describe how Berkmin decides which literal, out of the two possible literals of the already chosen variable, to pick. Berkmin maintains a counter *gcl*(*p*), measuring the global contribution of each literal to the conflicts. The counter *gcl*(*p*) is initialized to 0 and is increased whenever $cl'(p)$ is increased, but is not divided by a constant. If a topmost unsatisfied clause exists, Berkmin picks a literal with the highest global score *gcl*(*p*). Ties are broken randomly. If there is no unsatisfied topmost clause, then Berkmin picks the literal with the highest value of *two*(*p*), where *two*(*p*) approximates the number of binary clauses in the neighborhood of literal *p*. The function *two*(*p*) is computed as follows: First, the number of binary clauses containing *p* is calculated. Then, for each binary clause *B*, containing *p*, the number of binary clauses containing $\neg q$ is computed, where *q* is the other literal of *B*. The sum of all computed numbers gives the value of *two*(*p*). To reduce the amount of time spent computing *two*(*p*), a threshold value of 100 is used. As soon as the value of *two*(*p*) exceeds the threshold, its computation is stopped. Once again, ties are broken randomly.

The most important advantage of Berkmin's approach over VSIDS, as stated by the authors of Berkmin, is its additional dynamicity. It quickly adjusts itself to reflect changes in the set of variables relevant to the currently explored branch. Indeed, Berkmin picks variables from fresh conflict clauses and thus uses very recent data. Our understanding is that there is another important advantage of Berkmin's heuristic over VSIDS: newly assigned variables tend to embrace more interrelated variables. By "interrelated," we mean variables whose joint assignment increases the chances of both quickly reaching a conflict in an unsatisfiable branch and satisfying out "problematic" clauses in satisfiable branches. According to Berkmin's heuristic, a series of new decision variables appear in recent conflict clauses. Hence, these variables were recently traversed during conflict analysis and consequently contributed to conflict derivation. Moreover, even if the top-most conflict clauses were recorded a long time ago, the fact that their variables appeared closely together during conflict analysis, hints that they are interrelated. However, the impact of this advantage is diluted by the fact that Berkmin does not put the initial clauses on the list, but instead uses VSIDS as a secondary heuristic. The novel CBH heuristic, described in the next section, takes advantage of this observation.

5 The Clause-Based Heuristic

In our clause-based heuristic (CBH), all clauses (both the initial and the conflict clauses) are organized in a list. After each conflict, the conflict clause is prepended to the top of the list. *Conflict-responsible* clauses, that is, clauses visited during 1UIP conflict-clause identification, are placed just after the new conflict clause. The next decision literal is picked from the topmost unsatisfied clause of the list. One can see that CBH is highly dynamic, since recently visited clauses are placed at the top of the list. Also, CBH organizes the list in such way that clauses that were responsible for a recent conflict are placed together. Hence, when one picks a series of decision variables after backtracking, it will tend to embrace interrelated variables. Indeed, when literals are picked from the same clause they must be related, even if the clause is an initial clause. When literals are picked from closely-placed clauses, they also tend to be related, since the list is organized in such a way that interrelated clauses are near each other, by placing conflict clauses at the top and moving conflict-responsible clauses towards the top.

As a variant, CBH can also move clauses found to have exactly two unassigned literals during BCP to the top of the list. We refer to this strategy as *2LitFirst*. The added value of this strategy is that: (1) more implications are learned during BCP; (2) short and potentially contradictory clauses tend to be immediately satisfied. The first point guides the solver to find conflicts in an unsatisfiable area, and the second one is useful to eliminate conflicts in a satisfiable area. The disadvantages of *2LitFirst* are that: (1) it tends to separate between clauses that contain interrelated variables; (2) it may promote clauses that have never been responsible for conflicts.

We found experimentally that while usually *2LitFirst* hurts performance, it may be helpful for instances having high clause/variable ratio. This can be explained by the fact that, in instances having a high clause/variable ratio, variables tend to appear in a greater number of clauses, since there are fewer variables per clause overall. Hence, two chains of decisions taken using different decision strategies tend to contain more common variables. This gives more weight to the order between variables and the local context of the search. One should prefer variables whose assignment can have an immediate impact; this is exactly what *2LitFirst* does. The default version of CBH invokes *2LitFirst* on instances where the clause/variable ratio exceeds 10.

One can see that the major differences of CBH comparing with Berkmin's heuristic are the following:

(1) Both the initial and the conflict clauses, rather than only conflict clauses, are organized in a list; therefore, a second choice heuristic is not required. Thus, any set of decision variables picked by CBH tends to contain more variables from the same clause.

(2) After a conflict, in addition to the conflict clause, a number of clauses responsible for the conflict (including initial clauses) are moved towards the head of the list. Thus, clauses that are placed nearby are likely interrelated.

(3) As a variant, clauses that were discovered to have two unassigned literals are moved towards the top of the list.

CBH can be easily implemented using a doubly-linked list. A pointer to the currently watched clause *C*, initialized to the top-most clause, is maintained. When a decision is required, we seek the top-most unsatisfied clause *A,* starting from *C* towards the bottom of the list, and pick a literal from *A* (as described in Section 5.1). Observe that if no top-most unsatisfied clause exists, then we have a satisfying assignment, since all the clauses, including the original ones are satisfied. After each conflict, the solver updates the clause list and sets the currently watched clause to point to the top of the list.

The next subsection explains how CBH chooses the decision literal from the top-most unsatisfiable clause. Subsection 5.2 explains the initial organization of the clause list. Subsection 5.3 describes conflict-responsible clause identification in greater detail.

5.1 Choosing the Decision Literal from the Top-Most Clause

CBH maintains two counters, $lcl(p)$ and $gcl(p)$, measuring the local and global contributions of each literal to the conflicts, respectively. The counter $lcl(p)$ is initialized to 0 for each p, while $gcl(p)$ is initialized with the number of p's appearances in initial clauses. Both counters are incremented whenever a literal belongs to one of the clauses traversed in the implication graph during 1UIP conflict-clause identification. Occasionally, the value of $lcl(p)$ is divided by 2.

CBH also maintains two counters for variables $lcv(p)$ and $gcv(p)$, measuring the contribution of each variable to the conflicts. We define:

$$lcv(p) = (lcl\,(p) + lcl\,(\neg p)) + 3 \cdot min(lcl\,(p), lcl\,(\neg p)). \tag{2}$$

The first term gives preference to variables for which both literals are important, and the second term eliminates variables where only one literal is important. In a similar manner, we have:

$$gcv(p) = (gcl\,(p) + gcl\,(\neg p)) + 3 \cdot min(gcl\,(p), gcl\,(\neg p)). \tag{3}$$

CBH chooses the decision variable from the topmost unsatisfied clause using the following algorithm: A variable *p* with maximal *lcv*(*p*) is chosen, so as to give preference to variables that participated in recent conflicts. Ties are broken by preferring variables with maximal global score *gcv*(*p*). According to the next criterion, variables that used to have the maximal decision level when assigned the last time are preferred. (If there still is a tie, it is broken by picking the lexicographically smallest variable.)

CBH chooses the decision literal out of the two possible, based on the global contribution value *gcl*(*p*).

5.2 Initial Clause-List Organization

In general, we aim to:

(1) place clauses containing frequently appearing literals near the top of the list, and
(2) place clauses containing common literals nearby.

Point (1) guides the solver to start the search using frequent literals, and the second point increases the chances of picking interrelated literals.

First, the initial global score *igs*(*p*) is calculated for each literal *p*. The function *igs*(*p*) is initialized to 0 and is augmented for each clause that contains the literal *p*. The initial global score reflects the overall frequency of a literal.

In the process of clause list construction, we also maintain the initial local score *ils*(*p*) for each literal *p*. It is calculated similarly to *igs*(*p*), except that only clauses that were already placed on the clause list are considered. The local score reflects the involvement of *p* in clauses that have been already appended to the clause list. Initially, no clauses are included in the clause list, hence *ils*(*p*)=0 for each literal *p*. We also define the initial overall score *ios*(*p*) = *igs*(*p*) + *ils*(*p*) for each literal *p*. The initial overall score takes into consideration both the local and global influence of each literal.

So far, we have defined three functions for each literal reflecting its global, local and overall influence. Now, we can define the initial variable overall score for each variable *p*:

$$iosv(p) = (ios(p) + ios(\neg p)) + 3 \cdot \min(ios(p), ios(\neg p)). \tag{4}$$

The clause list is constructed by repeating the following procedure until all the clauses are placed on the clause list: Let *p* be the variable having the maximal variable overall score amongst all variables that have not already been picked. (Ties are broken by preferring the smaller variable according to lexicographical order.) We append clauses containing the variable *p* that have not yet been appended to the end of the clause list. Local and overall scores are updated for each literal participating in clauses that have been appended to the list. Such dynamic update of scores does not require any overhead, given that we use a priority queue, indexed by the scores. Literals can be moved within the queues in constant time.

5.3 Conflict-Responsible Clauses

Recall that after a conflict, conflict-responsible clauses are placed at the head of the list, after the new conflict clause. A clause is responsible for a conflict (in the context of CBH) if it appears in the implication clause of either a literal that appears on the conflict side of the 1UIP cut, or else of a literal whose negation appears in the 1UIP conflict clause.

Consider the example in Fig. 1. Clauses A_4, A_6, A_2 and A_5 are responsible for the conflict. Indeed, clauses A_4 and A_6 are the implication clauses of the two literals x and $\neg x$ that appear on the conflict side of the 1UIP cut, and clauses A_2 and A_5 are the implication clauses of the literals q and t, whose negation appear in the conflict clause.

From a practical point of view, it is not hard to identify clauses that are responsible for the conflict, since these are all the clauses that were visited during 1UIP conflict-clause construction.

6 Experimental Results

We implemented CBH within two SAT solvers. The first is the newest version of the famous zChaff solver, namely zchaff_2004.11.15 [10]. zChaff won first place in the Industrial-Overall category of the SAT'04 competition [9]. This new version of zChaff implements Berkmin's heuristic, in contrast with the old version of zChaff [13] which used VSIDS. The performance of zChaff was measured on a machine with 4Gb of memory and two Intel® Xeon™ CPU 3.06GHz processors with hyper-threading. The second solver we used in our experiments is SE—a Chaff-like solver, implementing 1UIP conflict-clause recording [13], non-chronological backtracking [12], frequent search restarts [5,6,10] and aggressive clause deletion [2,5,10] strategies. We designed SE to implement the aforementioned enhancements of DPLL, since they play a crucial role for the performance of a modern SAT solver, tuned for industrial benchmarks, as explained in papers on Berkmin [5] and the newest zChaff [10]. We did not check whether CBH boosts performance when one or more of the above mentioned techniques is not used. The performance of SE was measured on a stronger machine with 4Gb of memory and two Intel® Xeon™ CPU 3.20GHz processors with hyper-threading.

In what follows, we first analyze the overall performance of CBH versus Berkmin's heuristic, VSIDS and Berkmin-like new zChaff's heuristic. We show that CBH outperforms both Berkmin's heuristic and VSIDS within the SE SAT solver, and also that CBH significantly outperforms zChaff's new Berkmin-like heuristic. Then, we analyze how various strategies used by CBH contribute to its performance. We tested CBH inside two solvers to ensure that its measured impact on performance is independent of the implementation details of a particular solver. The main measure for success in our experiments is the number of solved instances within an hour on hard industrial families used during SAT'04 competition [9]. We find this measure, which was used during the SAT'04 competition, more convincing than a comparison of the number of decisions or conflicts (omitted for lack of space), since reducing the running time is the final goal of any practical heuristic. Our experiments required approximately 35 days of computation.

Table 2. Description of the hard industrial benchmark families used in our experiments. Family name, number of instances in each family as well as the average, and maximal and minimal clause/variable ratios are provided

Abbreviation	*Family Name*	*Inst. Num.*	*Cls/Var Avrg*	*Cls/Var Max*	*Cls/Var Min*
HEQ	goldberg03-hard_eq_check	13	6.4	6.7	6.1
GR	maris03-gripper	10	9.1	9.7	8.5
SCH	schuppan03-l2s	11	3.2	3.3	3.0
ST2	simon03-sat02	9	3.3	4.2	2.7
ST2B	simon03-sat02bis	10	23.9	71.9	2.9
CLR	vangelder-cnf-color	12	42.9	195.4	4.0
PST	velev-pipe-sat-1-1	10	33.7	33.8	33.7
VUN	velev-vliw_unsat_2.0	8	15.6	20.1	10.7

Table 3. Performance of CBH vs. two versions of VSIDS and Berkmin's heuristic, implemented in the SE SAT solver, on eight hard industrial families. The first column contains an abbreviated family name. Each pair of subsequent columns is dedicated to a specific heuristic,. The number of instances, solved within an hour, is provided

Family	*CBH solved*	*Berkmin heur.*	*VSIDSM*	*VSIDS*
HEQ	5	4	4	3
GR	1	1	0	1
SCH	5	2	2	0
ST2	5	4	4	2
ST2B	2	2	2	1
CLR	6	4	4	4
PST	10	10	4	5
VUN	4	2	1	0
ALL	***38***	***29***	***21***	***16***

Table 3 compares the performance of CBH, VSIDS, VSIDSM—MiniSat-like VSIDS with frequent scores decay and Berkmin's heuristic, implemented in SE, on eight hard industrial families used during the SAT'04 competition (downloaded from the competition's website [16]). The description of these families is provided in Table 2. VSIDSM multiplies the score by 0.95 after every 10 conflicts, rather than with each conflict. The latter rate is used within MiniSat, but the former is preferable within SE. Other heuristics decay the scores each 6000 conflicts. CBH solves at least as many instances within each family, when compared to either Berkmin's heuristic or either version of VSIDS. CBH solves more instances than both version of VSIDS in 7 out of 8 cases, and solves more instances than Berkmin's heuristic in 5 out of 8 cases.

Table 4 shows the performance of CBH within the new version of zChaff. The performance of a version of CBH that does not use *2LitFirst* is also provided. One can see that zChaff, enabled with CBH, outperforms zChaff in a very convincing manner for 6 out of 8 families and is inferior in only once case. Moreover, if the *2LitFirst* strategy is not used, CBH is never inferior.

Table 4. CBH vs. the default heuristic within zChaff_2004.11.15. CBH_2L_N is a version of CBH that does not use the *2LitFirst* strategy

Family	*zChaff + CBH*	*zChaff+ CBH_2L_N*	*zChaff default*
HEQ	8	8	4
GR	3	3	0
SCH	5	5	2
ST2	4	4	1
ST2B	2	2	1
CLR	3	4	1
PST	5	8	8
VUN	2	2	2
ALL	***32***	***36***	***19***

One can conclude that CBH definitely contributes to modern SAT-solver performance, outperforming both VSIDS and Berkmin's heuristic. Our experiments also confirm that Berkmin's heuristic is preferable to VSIDS, though the gap narrows if VSIDS uses frequent scores decay. This is to be expected, but—to the best of our knowledge—has never been reported, despite the fact that Berkmin's heuristic has been partially or fully adopted by the most modern SAT solvers [1,4,5,10].

What remains is to analyze the performance of CBH when disabling some of its specific strategies. Accordingly, we consider CBH_NM, a version that does not move conflict-responsible clauses to the top of the list (but still appends the conflict clause itself to the top of the list), and CBH_NI, which does not use the initial strategy, described in Section 5.2, but rather appends all clauses to the list in the order of appearance in the input instance. We also experimented with CBH_2L_A, which always uses *2LitFirst*, and with CBH_2L_N, which never does. Table 5 and 6 compare the performance of CBH within SE and zChaff respectively.

Switching off the initial strategy results in a performance degradation for 3 families within SE, and in performance gain for one family. In zChaff, switching off the initial strategy resulted in performance degradation for 2 families. In general, the initial strategy improves the performance within both SE and zChaff, although it is not the most crucial factor contributing to CBH's performance. Even if the initial strategy is switched off, CBH performs better than other decision heuristics. This can be explained by the fact that during the search, CBH quickly reorganizes the clause list to contain groups of interrelated clauses.

Switching off the moving of conflict-responsible clauses to the top of the list results in a performance degradation for 4 families and performance gain on 3 families in zChaff. The overall number of solved instances is higher when the strategy is switched on. Switching off the moving of conflict-responsible clauses to the top of the list leads to mixed results in the case of SE. Performance seriously degrades for the SCH family, and also for the ST2 family; however, there is a performance-boost for the GR, ST2B and VUN families. The overall number of solved instances remains the same. One can conclude that moving the conflict-responsible clauses to the top of the list can be useful for some families, but hurt

Table 5. Performance of different configurations of CBH in terms of solved instances within an hour in SE solver

Family	*CBH*	*CBH_NM*	*CBH_NI*	*CBH_2L_A*	*CBH_2L_N*
HEQ	5	4	5	3	5
GR	1	2	2	0	1
SCH	5	2	4	5	5
ST2	5	4	5	2	5
ST2B	2	3	1	2	2
CLR	6	7	5	5	6
PST	10	10	10	10	10
VUN	4	6	4	4	1
ALL	***38***	***38***	***36***	***31***	***35***

Table 6. Performance of different configurations of CBH in terms of solved instances within an hour in zChaff solver

Family	*CBH*	*CBH_NM*	*CBH_NI*	*CBH_2L_A*	*CBH_2L_N*
HEQ	8	4	8	4	8
GR	3	1	3	0	3
SCH	5	2	4	0	5
ST2	4	2	4	0	4
ST2B	2	3	2	2	2
CLR	3	3	3	3	4
PST	5	10	3	5	8
VUN	2	3	2	2	2
ALL	***32***	***28***	***29***	***16***	***36***

others. We recommend invokeing it by default, since it results in an overall performance boost in the case of zChaff and does not hurt overall performance of SE.

Regarding the impact of *2LitFirst* strategy, first observe that according to Tables 5 and 6, invoking *2LitFirst* on every instance does not pay off. Note that the default strategy used by CBH invokes *2LitFirst* if the clause/variable rate is greater than 10. The motivating experimental observation for designing CBH in this manner is that SE without *2LitFirst* performs the same as the default version on all families, except VUN, where performance seriously degrades. VUN is the only family, other than PST, for which the clause/variable ratio is greater than 10 for all instances. To confirm that SE performs better when *2LitFirst* is invoked on instances with high clause/variable ratio, we launched SE on 26 handmade families that were submitted to SAT'04 (and are available at [16]). SE was able to solve at least 1 instance from 13 families. The default CBH strategy performed better than a strategy with *2LitFirst* disabled on 2 out of 13 families, and it performed the same on other families. In the case of zChaff, we found that *2LitFirst* invocation hurts performance in a dramatic manner on families with low clause/variable ratio, and leaves the performance the same or slightly degraded on families having high clause/variable ratio. The families HEQ, GR, SCH and ST2 are those with a ratio for all its instances lower than 10. If *2LitFirst* is always used, zChaff is able to solve only 4 instances of these 4 families, comparing to 20 instances solved by the version that never uses *2LitFirst*. The other 4 families have either mixed or high clause/variable ratio. On these families, when *2LitFirst* is always used, zChaff is able to solve 12 instances, compared to 16 by a version that never uses *2LitFirst*. One can conclude that within zChaff, *2LitFirst* usage is not justified. Overall, *2LitFirst* performs much better on instances having a high clause/variable ratio.

7 Conclusions

We have presented a novel clause-based heuristic, CBH. It maintains a clause list organized in a manner that allows the algorithm to choose sequences of interrelated variables that were responsible for recent conflict derivation.

CBH maintains both the initial and the conflict clauses in a single list. The next decision literal is picked from the topmost unsatisfied clause of the list. After each conflict, the conflict clause is prepended to the top of the list. Clauses visited during conflict-clause identification, are placed just after the new conflict clause. As a variant, if the clause/variable ratio of the input instance is greater than a predefined value (10 is a reasonable choice), newly identified binary clauses are moved to the top of the list.

We have demonstrated that using CBH results in a significant performance boost on hard industrial families, when compared with Berkmin's heuristic or VSIDS.

Acknowledgements

The work of the third author was carried out in partial fulfillment of the requirements for a Ph.D. We thank Ranan Fraer for his careful reading and helpful comments.

References

[1] J. Alfredsson. The SAT solver Oepir. URL http://www.lri.fr/~simon/contest/results/ONLINEBOOKLET/OepirA.ps (viewed January, 16 2005).

[2] R. Bayardo, Jr., and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In Proceedings of the National Conference on Artificial Intelligence, pp. 203-208, 1997.

[3] M. Davis, G. Logemann and D. Loveland. A machine program for theorem proving. In Communications of the ACM, (5): 394-397, 1962.

[4] N. Eén and N. Sörensson. An extensible SAT-solver. In Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003), May 2003.

[5] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In Design, Automation, and Test in Europe (DATE '02), p. 142-149, March 2002.

[6] C.P. Gomes, B. Selman and H. Kautz. Boosting combinatorial search through randomization. In Proceedings of the National Conference on Artificial Intelligence, July 1998.

[7] R.G. Jeroslaw and J. Wang. Solving propositional satisfiability problems. In Annals of mathematics and Artificial Intelligence, (1):167-187, 1990.

[8] H. Kautz, B. Selman. Planning as satisfiability. In Proceedings of the 10th European conference on Artificial intelligence, 1992.

[9] D. Le Berre and L. Simon. Fifty-five solvers in Vancouver: The SAT 2004 competition. In Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004), volume Lecture Notes in Computer Science, 2004. Accepted for publication.

[10] Y.S. Mahajan, Z. Fu, S. Malik. ZChaff2004: an efficient SAT solver. In Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004), volume Lecture Notes in Computer Science, 2004. Accepted for publication.

[11] J.P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA), September 1999.

[12] J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers, (48):506-521, 1999.

[13] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik. Chaff: Engineering an efficient SAT solver. In Proceedings of the Design Automation Conference, 2001.

[14] M. R. Prasad, A. Biere, A. Gupta. A survey of recent advances in SAT-based formal verification, Intl. Journal on Software Tools for Technology Transfer (STTT), (7):156-173, January 2005.

[15] L. Ryan, Efficient algorithms for clause-learning SAT solvers. Masters thesis, Simon Fraser University, February 2004.

[16] Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004), May 2004. URL http://satlive.org/SATCompetition/2004/ (viewed September 1, 2004).

[17] R. Zabih and D.A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In Proceedings of National Conference on Artificial Intelligence, p. 155-160, 1988.

[18] L. Zhang, C.F. Madigan, M.H. Moskewicz and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. International Conference on Computer-Aided Design (ICCAD'01), p. 279-285, November 2001.

Effective Preprocessing in SAT Through Variable and Clause Elimination

Niklas Eén and Armin Biere

Chalmers University of Technology,
Göteborg, Sweden
Johannes Kepler University,
Linz, Austria

Abstract. Preprocessing SAT instances can reduce their size considerably. We combine variable elimination with subsumption and self-subsuming resolution, and show that these techniques not only shrink the formula further than previous preprocessing efforts based on variable elimination, but also decrease runtime of SAT solvers substantially for typical industrial SAT problems. We discuss critical implementation details that make the reduction procedure fast enough to be practical.

1 Introduction

The size of CNF formulas is often very large, particularly in the context of formal verification. In theory, a very large formula may be easy to solve and a small formula hard. However, in practice, it is often observed that the runtime of a SAT solver is very much related to the size of the input formula, at least when the formulas stem from the same set of problems.

This paper presents new techniques which reduce the size of a CNF formula in order to speed up overall SAT solving time. Our experiments on problems from industrial circuit verification show large speedups, not only compared to plain SAT solving, but also compared to related preprocessing techniques [1].

Modern SAT solvers use unit propagation and the pure literal rule in a preprocessing phase, as already described in the original DPLL algorithm [2]. More sophisticated techniques focus on deriving units, implications and equivalent literals [3, 4, 5, 6, 7]. Other similar techniques have been used in the context of ATPG, such as *recursive learning* [8], or in circuit verification, in particular *and-inverter graphs* [9] and *BDD-sweeping* [9]. The latter techniques have in common that they allow to restructure circuits but do not directly apply to CNF. Our approach is orthogonal to these techniques in the sense that it can be applied in addition to restructuring, after a CNF has been produced. Furthermore, from the CNF, individual clauses can be removed, an operation without correspondence in the circuit representation.

Preprocessing in SAT is a trade-off between the amount of reduction achieved and invested time. Light weight approaches such as [5] focus on fast preprocessing. Their running time is usually negligible compared to the overall solving time.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 61–75, 2005.

On the other side of the spectrum lie techniques which in practice take considerable time. Examples are saturation of hyper binary resolution [3], k-saturation for $k \geq 2$ in Stålmarck's method [7], or saturation of recursive learning for larger recursion depths. These techniques can only be applied if a huge benefit is expected—which is application domain depended—or if a time limit is enforced.

Recently, the rule of *elimination of atomic formulas* from [10], which eliminates variables from a CNF by *clause distribution*, has been reconsidered as a basis for symbolic DPLL in the ZDD based SAT solver ZRes [11], as a way to eliminate variables in the QBF solver Quantor [12], and, independently in the preprocessor NiVER [1]. Clause distribution is a light weight preprocessing technique, as long as a limit on the growth of the clause data base is enforced.

We extend [1] by three new techniques: subsumption, self-subsuming resolution, and variable elimination by substitution. This results in much higher reduction rates and faster SAT solving, as we show in our experiments. The description of the implementation of NiVER [1] stays on a very high level. We describe two implementation techniques for speeding up the process, based on (1) restricting the set of variables considered for elimination to *touched* variables, and (2) using *signatures* for fast subsumption checks. The latter has has also been used in [12], but the focus there is QBF, and no experimental results for preprocessing SAT instances are given. The optimizations allow us to keep the runtime small enough for a light weight approach.

Finally, we believe that preprocessing and encoding are two sides of the same coin. One way of speeding up SAT solving is to work on sophisticated CNF encoding algorithms such as [13, 14, 15, 16]. We suggest, as an alternative, that the CNF is simplified after its generation, which is less application domain dependent. From a pragmatic point of view, it also eases the burden of developing good domain specific CNF encoders if the SAT solver is known to do a good job of reducing verbose CNF formulations. Furthermore, our simplification techniques are all resolution based and can therefore easily be incorporated in a solver with refutation generation. We leave it to future work to compare the two different approaches, particularly since the number of available CNF encoders and propositional non-CNF problems is currently rather small.

Generally, our simplification techniques can be applied in three different ways: (1) during preprocessing, (2) during SAT solving, e.g. at restart, or (3) between two incremental SAT problems. We will focus on applying simplification as a preprocessor, although a small study is included of an application in an incremental SAT problem.

To summarize, our contributions are the following. We extend NiVER [1] by subsumption, which was already discussed by one of the authors in the context of QBF [12]. Furthermore, we present two new techniques, self-subsuming resolution and variable elimination by substitution. Beside a compact description of the preprocessing algorithm itself we discuss two important low-level optimizations. Finally we show the effectiveness of these techniques as implemented in SatELite on a comprehensive set of industrial benchmarks.

2 Preliminaries

A CNF consists of a set of *clauses*, where each clause C is a set of literals. A literal is a boolean variable x or its negation $\overline{x}$.

Given two clauses $C_1 = \{x, a_1, \ldots, a_n\}$ and $C_2 = \{\overline{x}, b_1, \ldots, b_m\}$ the implied clause $C = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$ is called the *resolvent* of the two original clauses by performing *resolution* on the variable x. We write $C = C_1 \otimes C_2$. This notion can be lifted to sets of clauses. Let S_1 be a set of clauses which all contain x, S_2 a set of clauses which all contain $\overline{x}$. Then $S_1 \otimes S_2$ is defined as

$$S_1 \otimes S_2 = \{C_1 \otimes C_2 \mid C_1 \in S_1,\ C_2 \in S_2\}$$

The basic simplification technique in this paper, and also in NiVER [1], follows [10] and simply eliminates variables. In a given CNF, let S_x be the set of clauses in which x occurs, $S_{\overline{x}}$ be the set of clauses in which $\overline{x}$ occurs and define $S = S_x \cup S_{\overline{x}}$.

The elimination of a variable x in the whole CNF can be computed by pairwise resolving each clause in S_x with every clause in $S_{\overline{x}}$. The produced resolvents $S' = S_x \otimes S_{\overline{x}}$ replace the original clauses S containing x or $\overline{x}$, resulting in a satisfiability equivalent problem. We refer to this procedure as elimination by *clause distribution*, and count only non-trivial clauses as part of the result. A clause is trivial if it contains a variable and its negation.

In principle, the resolution operator $\otimes$ should have the resolution variable x as parameter. However, if clauses can be resolved with respect to different resolution variables, then all the resolvents will be trivial anyhow.

3 New Simplifications

In early experiments and also in the context of QBF [12] we observed that clause distribution produces many subsumed clauses. A clause C_1 is said to (syntactically) *subsume* C_2 if $C_1 \subseteq C_2$. A subsumed clause is redundant and can be discarded from the SAT problem. Particularly, a subsumed clause never needs to be part of a resolution proof of unsatisfiability.

We also observed that often similar clauses of a particular kind occur: one clause C_2 almost subsumes a clause C_1, except for one literal $\overline{x}$, which, occurs with the opposite sign in C_2. For instance, let $C_1 = \{x, a, b\}$, and $C_2 = \{\overline{x}, a\}$, then resolving on x will produce $C_1' = \{a, b\}$, which subsumes C_1. Thus after adding C_1' to the CNF, we can remove C_1, in essence eliminating one literal. In this case, we say that C_1 is *strengthened* by *self-subsumption* using C_2. This simplification rule is called *self-subsuming resolution.*

As we will show in the experimental section, adding these subsumption techniques to variable elimination through clause distribution gives huge benefits compared to [1].

If a circuit is encoded in CNF, typically using the Tseitin transformation [17], then many variables are actually *functionally dependent* on other variables, particularly those introduced for gate outputs. In previous work, this information has been used to restrict the set of decision variables in a SAT solver to functionally *independent* variables [18, 19]. We use the information to simplify the CNF, essentially extracting gates as in [19]. Output variables of gates are functionally dependent on input variables. If the following three clauses

$$\ldots \{x, \overline{a}, \overline{b}\}, \{\overline{x}, a\}, \{\overline{x}, b\} \ldots \tag{1}$$

are part of a CNF then the AND gate $x = (a \wedge b)$ can be extracted, showing that x is functionally dependent. We also call this equation a *definition* of x.

If x has a definition and is eliminated by clause distribution, many redundant resolvents are generated. By using the definition these clauses can be removed easily. Let G be the set of clauses used for extracting a *gate* with output x. Further recall that S_x is the set of clauses of S in which x occurs and similarly define G_x, and $G_{\overline{x}}$. Then the set S of all clauses with x or $\overline{x}$ can be partitioned into $S = G \cup R$, with $R \equiv S \backslash G$ the set of *remaining* clauses not used for extracting the gate. From $S = (G_x \cup R_x) \cup (G_{\overline{x}} \cup R_{\overline{x}})$ it follows that the set S' of all resolvents can be partitioned into $S' = S'' \cup G' \cup R'$ with

$$S'' = (R_x \otimes G_{\overline{x}}) \cup (G_x \otimes R_{\overline{x}}), \quad G' = G_x \otimes G_{\overline{x}}, \quad \text{and} \quad R' = R_x \otimes R_{\overline{x}}.$$

Furthermore, we have the following Theorem, which shows that S'' implies G' and R', allowing S' to be replaced by S''.

Theorem. $S'' \models G' \cup R'$

The proof follows by first noticing, as in [20], that G' contains only trivial clauses. All the resolvents in R' can be obtained through several resolution steps (linear in the width of the gate or by just one hyper resolution step [3]) from clauses in S''. Another view is to substitute in R all occurrences of x by its definition (x by $a \wedge b$ and $\overline{x}$ by $\overline{a \wedge b}$ in the example) and then apply the distributivity law to obtain a flat CNF.

As a result, in the elimination of a functional dependent variable the clauses in G' and R' do not have to be added, which always reduces the number of added resolvents. We call this simplification rule *variable elimination by substitution.* To continue the example in Eqn. (1) , let S be

$$\underbrace{\overset{1}{\{x, c\}}, \overset{2}{\{x, \overline{d}\}}}_{R_x}, \underbrace{\overset{3}{\{x, \overline{a}, \overline{b}\}}}_{G_x}, \underbrace{\overset{4}{\{\overline{x}, a\}}, \overset{5}{\{\overline{x}, b\}}}_{G_{\overline{x}}}, \underbrace{\overset{6}{\{\overline{x}, \overline{e}, f\}}}_{R_{\overline{x}}}$$

The resolvents are:

$$\overset{1\otimes 4}{\{c, a\}}, \overset{1\otimes 5}{\{c, b\}}, \overset{2\otimes 4}{\{\overline{d}, a\}}, \overset{2\otimes 5}{\{\overline{d}, b\}}, \overset{3\otimes 6}{\{\overline{a}, \overline{b}, \overline{e}, f\}} \ (S'')$$

$$\overset{3\otimes 4}{\{\overline{a}, \overline{b}, a\}}, \overset{3\otimes 5}{\{\overline{a}, \overline{b}, b\}} \ (G') \qquad \overset{1\otimes 6}{\{c, \overline{e}, f\}}, \overset{2\otimes 6}{\{\overline{d}, \overline{e}, f\}} \ (R')$$

G' has only trivial clauses. Since trivial clauses are not counted, we have $|S'| = 7 > 5 = |S''|$. Replacing S with S'' results in a decrease of the number of clauses from 6 to 5, while the full clause distribution actually results in an increase from 6 to 7. Also note that the redundant clauses in R' can be obtained from S''

through two resolution steps each (actually by one hyper resolution step [3]): $1{\otimes}6 = (1{\otimes}4) \otimes ((1{\otimes}5) \otimes (3{\otimes}6))$ and $2{\otimes}6 = (2{\otimes}4) \otimes ((2{\otimes}5) \otimes (3{\otimes}6))$.

We also realized that subsumption sometimes removes clauses which could be used to extract a gate. For instance if the clause $C = \{\overline{a}, \overline{b}\}$ is added to the CNF in Eqn. (1), then the clause $\{x, \overline{a}, \overline{b}\}$ is removed and no AND gate can be extracted anymore. However, by one hyper resolution step, or two ordinary resolution steps, of C with the original two binary clauses the unit x can be derived, which, of course, simplifies the CNF even further. For all clauses C, we try to find binary clauses, that, if resolved with C in one hyper resolution step produce a unit. We call this simplification rule *hyper-unary-resolution*, similar to hyper-binary-resolution of [3].

4 Implementation

We present an implementation that should work for any clause based SAT solver, including those with an incremental SAT interface. In that context, the simplification can be applied between the different incremental SAT instances.

The techniques in this paper aim at simplifying a SAT problem by reducing its size. Variable elimination is applied greedily until no more improvement can be made to the clause database by a single elimination. Different notions of "improvement" can be used, and previous work [1] is focused on minimizing the number of *literal occurrences*. In our implementation we minimize the number of *clauses*. The rationale behind this is that propagation in a SAT solver is roughly proportional to the number of clauses, independent of their size.

4.1 Touched-Lists

Subsumption and variable elimination interact, such that strengthening or removing a clause by (self-) subsumption can turn the elimination of a variable into an improvement, and eliminating a variable, which produces new clauses, might give new opportunities for subsumption.

In our implementation, subsumption and elimination are alternated until a fixed-point is reached. To make this efficient, it is important not to loop repeatedly over all clauses. Therefore, three sets are maintained, storing information about the modifications made to the clause database:

Touched (set of variables). A variable is added to this set if it occurs in a clause being added, removed, or strengthened. Initially all variables are "touched".

Added (set of clauses). When a clause is added to the SAT problem (e.g. by variable elimination), it is also added to this set. Initially all clauses are considered "added".

Strengthened (set of clauses). When a clause is strengthened (one literal is removed, either by self-subsumption or toplevel propagation[1]) it is added to this set. Initially the set is empty.

[1] Unit propagation performed under no assumptions, as opposed to during the search.

These sets are repeatedly *cleared* during the simplification procedure described in Sect. 4.3, then populated again as new clauses are produced during variable elimination, and while existing clauses are removed or strengthened by subsumption and self-subsumption. The algorithm terminates with all sets empty. In an incremental context—although not the focus of this paper—we note that new clauses can be added between SAT problems, populating *Added*, and that unit facts learned during the solving of one incremental SAT instance might remove or strengthen clauses, populating *Touched* and *Strengthened*.

4.2 Subsumption

The efficiency of subsumption is most important and is achieved by two implementation techniques. First, for each clause a 64-bit *signature* is stored [12]. The signature abstracts the set of literals of a clause in the following way: A hash function h maps literals to numbers 0..63, and the signature of a clause C is calculated as the bitwise Or of $2^{h(p)}$ over its literals $p \in C$. Then for each literal an *occur* list is maintained, pointing to all the clauses in which the literal occurs.

Now, *backward* subsumption, that is checking if a clause *subsumes* (as opposed to being *subsumed by*) some other clause in the database, can be implemented as follows:[2]

```
findSubsumed(Clause $C$)
  pick the literal $p$ in $C$ with the shortest occur list
  for each $C' \in occur(p)$ do
      if ($C \neq C'$ && $size(C) \leq size(C')$ && $subset(C, C')$)
          add $C'$ to result
  return result
subset(Clause $C$, Clause $C'$)
  if ($sig(C)$ & ~$sig(C') \neq 0$) return False
  else return result of iterating over $C$ and $C'$ in a
                   complete (expensive) subset test
```

This algorithm is very fast and allows backward subsumption to be applied *eagerly* to each added or strengthened clause. We rely on this fact in Sect. 4.3. Given a procedure for finding subsumed clauses, we can now define a method for using a clause C to strengthen other clauses by self-subsumption:

```
selfSubsume(Clause $C$)
  for each $p \in C$ do
      for each $C'$ subsumed by $C[p := \overline{p}]$ do
          $strengthen(C', \overline{p})$                    – remove $\overline{p}$ from $C'$
```

[2] && denotes logical And, & bitwise And, and ~ bitwise negation.

For the clause $\{a, b, c\}$ this method would call *findSubsumed*() for $\{\overline{a}, b, c\}$, $\{a, \overline{b}, c\}$, $\{a, b, \overline{c}\}$, and strengthen any result returned. It should be noted that the order of strengthening matters, but is not optimized in our implementation.

4.3 The Toplevel Simplification Method

We now state the main algorithm. The *post-conditions* are: (1) No opportunities remain for subsumption or self-subsumption. (2) No improvement can be made by eliminating a variable, unless the heuristic cut-off is used (see below). (3) The three sets *Touched*, *Added*, and *Strengthened* are empty.

```
simplify()
  do
      – SUBSUMPTION:
      $S_0$ = {set of clauses containing a literal occurring in
              some clause in Added}
      do
          $S_1$ = {set of clauses containing a literal occurring
                  negatively in some clause in Added}
              ∪ Added ∪ Strengthened
          clear Added and Strengthened
          for each $C \in S_1$ do selfSubsume($C$)
          propagateToplevel()        – may strengthen/remove clauses
      while (Strengthened $\neq \emptyset$)
      for each $C \in S_0$ not deleted do subsume($C$)

      – VARIABLE ELIMINATION:
      do
          $S$ = Touched ; clear Touched
          for each $x \in S$ do maybeEliminate($x$)
                      – eliminating variables will touch other variables
      while (Touched $\neq \emptyset$)
  while (Added $\neq \emptyset$)
```

The method *subsume*(C) removes any clause subsumed by C, and similarly *selfSubsume*(C) removes a literal from any clause that may be strengthened using C. The method *maybeEliminate*(x) removes x by clause distribution or substitution if the number of clauses is reduced. Finally, *propagateToplevel*() removes any satisfied clause or false literal permanently from the clause database, assigning variables and repeating the process if unit clauses are produced.

In the subsumption phase, two sets are computed: S_0 for standard subsumption, and S_1 for self-subsumption. Self-subsumption is applied first as it may render more (standard) subsumptions possible.

Because backward subsumption is eagerly applied to all added or strengthened clauses, the only candidates for being *subsumed* are the clauses of *Added*. Strengthened clauses cannot be subsumed as they now have fewer literals and were not subsumed before strengthening. A necessary condition for C to subsume C' is that C has at least one literal in common with C'. This motivates the definition of S_0.

Let "original clause" denote a clause not in *Added* or *Strengthened*. For self-subsumption (the set S_1) any added or strengthened clause can be used to remove literals from an original clause. Original clauses may self-subsume added clauses. This does not apply to strengthened clauses, since they have already been checked while still containing more literals. It remains to add to S_1 the original clauses that may strengthen a clause in *Added*. All the candidate clauses have to contain one literal $\overline{p}$ for some p in the added clauses.

4.4 Variable Elimination

The variable elimination procedure relies on three readily implemented methods, which we state here without pseudo-code:

maybeClauseDistribute(x) eliminates x by clause distribution if the result has fewer clauses than the original (after removing trivially satisfied clauses).

findDefinition(x) returns either $x \leftrightarrow p_1 \vee p_2 \vee \ldots \vee p_n$ or $x \leftrightarrow p_1 \wedge p_2 \wedge \ldots \wedge p_n$ or NoDef. Unit information is also detected by hyper-unary-resolution and returned as $x \leftrightarrow$ True or $x \leftrightarrow$ False. Note that in general there may be many definitions. We use the shortest one and do not extract further information from this.

maybeSubstitute(*def*) takes the definition of a functionally dependent variable and substitutes each occurrence of the variable by its definition, provided this results in fewer clauses. Substituting a literal by a disjunction is unproblematic; substituting by a conjunction requires duplicating the destination clause for each literal of the conjunction, as explained in Sect. 3.

```
maybeEliminate( Var $x$)
  if ($x$ assigned or has zero occurrences) return
  if (#occurs of $x$ and $\overline{x}$ are both $> 10$) return – heuristic cut-off

  def = findDefinition($x$)
  if (def $\neq$ NoDef) maybeSubstitute(def)
  else                  maybeClauseDistribute($x$)

  if ($x$ was eliminated)
      propagateToplevel()
      remove learned clauses with $x$       – for incremental SAT only
```

It was observed in an early implementation of the simplification procedure that on some problems the majority of time was spent on failed attempts to eliminate variables occurring frequently in both polarities. This is why these variables are heuristically excluded. The last line of the pseudo-code is only relevant in an incremental context; if variable elimination is applied during preprocessing, no learned clauses will exist.

4.5 Variable Elimination Related Issues

The elimination of variables results in a partial model if the problem is satisfiable. Clauses removed during variable elimination must therefore be stored and used to complete the model, if the full model is not needed. If not, removed clauses can simply be discarded.

Variable elimination also causes problems for the incremental SAT interface. Later extensions of the SAT instance might reintroduce eliminated variables, rendering the elimination unsound. Bringing back the removed clauses will solve the problem, but a simpler solution is to extend the solver interface to let the user explicitly prevent the elimination of selected variables.

5 Experimental Results

The techniques presented in this paper were implemented in a tool SatELite. It is downloadable together with the benchmarks and the result files used to produce the tables and diagrams of this section.[3] Three SAT solvers were used in our evaluation: (1) MiniSat v1.13 [21] with an improved conflict clause analysis [22]; (2) ZChaff version "Chaff II"; and (3) BerkMin v5.61. The benchmarks were selected to be relevant for *circuit verification*. To get a relevant measure for the reduction achieved by our preprocessing techniques, unit clauses were removed by performing a toplevel propagation using MiniSat prior to benchmarking.

For our evaluation, two benchmark sets were created. The first set, referred to as "IBM Problems", is a subset of the huge BMC benchmark set made available by E. Zarpas at IBM.[4] The benchmark set is divided into directories, each containing BMC problems of different lengths generated from the same circuit with the same specification. Without any prior knowledge of the benchmarks, we randomly selected a subset of the directories resulting in 355 problems.

The second set, referred to as "Industrial Mix" contains a mix of hardware verification problems, obtained as follows: The available industrial problems of the SAT-2004 Competition were downloaded. Problems concerning graph coloring, set covering and planning problems were removed. Our focus is on circuit verification. We also removed *Miroslav Velev's* problems because SatELite ran out of memory on some of them, which complicated benchmarking.[5] However, we note that the problems are already clausified in a smart way [16], which leaves little room for improvement by our methods. For the CNFs which SatELite could preprocess, reduction rates of less than 5% were achieved, and no measurable speedup. This supports our hypothesis that our method is an *alternative* to producing optimized CNFs directly from the source problem.

[3] www.cs.chalmers.se/~een/SatELite

[4] www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html

[5] The occurrence lists necessary for our preprocessing double the memory footprint. Although this is not a big issue, Velev's problems are among the largest that today's SAT solvers can handle. The current version of SatELite has not been optimized for memory performance.

Table 1. *Size-reduction comparison with* NiVER. "*v*", "*c*", "*l*" denote the number of variables, clauses, and literals in thousands respectivly. Times "*t*" are in seconds as provided by the Unix command "time", and include parsing and writing the result file. "SatELite as NiVER" uses no subsumption and has the same heuristic as NiVER for variable elimination (enforce fewer literals). It shows that SatELite can mimic NiVER well, and that our implementation techniques runs faster. The last column shows SatELite with all reductions on, which results in a strict improvement in size

	Original			NiVER					SatELite as NiVER					Full SatELite				
Name	*v*	*c*	*l*	*v*	*c*	*l*	:	*t*	*v*	*c*	*l*	:	*t*	*v*	*c*	*l*	:	*t*
6pipe	16	395	1157	15	393	1155	:	4.4	15	393	1155	:	2.2	**12**	**323**	**1018**	:	53.0
abp1-1-k31	15	48	124	8	34	98	:	0.6	8	33	94	:	0.3	**3**	**18**	**63**	:	1.2
barrel9	9	37	102	4	21	66	:	0.5	4	20	**65**	:	0.5	**2**	**16**	87	:	3.3
cache_10	227	880	2192	130	606	1680	:	20.6	92	417	1146	:	7.9	**29**	**178**	**748**	:	58.6
comb2	32	112	274	20	89	231	:	1.6	20	89	231	:	0.8	**3**	**18**	**63**	:	4.4
f2clk_40	28	80	186	10	44	125	:	1.4	7	32	90	:	0.5	**4**	**25**	**81**	:	1.2
fifo8_400	260	708	1602	69	301	859	:	13.6	42	164	451	:	6.5	**23**	**129**	**446**	:	11.2
guid-1-k56	99	307	758	45	193	553	:	3.9	44	189	540	:	3.1	**23**	**130**	**443**	:	8.0
ibm-03_k80	89	375	973	56	308	887	:	5.5	44	230	661	:	1.9	**28**	**190**	**629**	:	5.8
ibm-20_k45	91	373	945	46	281	832	:	6.7	41	250	725	:	2.1	**20**	**156**	**546**	:	7.0
ip50	66	215	513	34	148	398	:	5.1	12	50	**134**	:	1.6	**8**	**43**	139	:	4.2
longmult15	8	24	59	4	16	46	:	0.3	3	14	39	:	0.1	**1**	**9**	**28**	:	0.4
w08_14	120	425	1038	69	324	859	:	7.2	69	324	856	:	3.7	**34**	**220**	**688**	:	15.7

Finally, we added 18 satisfiable and 18 unsatisfiable BMC problems used in [23], mainly from the *Texas'97 benchmarks*;[6] 18 unsatisfiable BMC problems generated from the *PicoJava* design[7] and 13 liveness problems from *SatLib*.[8] The result contains 115 CNF files.

***Study 1 – Comparing reduction rates with* NiVER.** This study shows that SatELite is an improvement over earlier work. We use the same problem set as presented in the NiVER paper [1]. The results are shown in *Table 1.*

Study 2 – Reduction rates and preprocessing time. *Figure 1* shows the effect of applying preprocessing in terms of the number of remaining variables, clauses, and literal occurrences. We see that for most problems the number of clauses drop significantly, as well as the number of literal occurrences (with some exceptions), resulting in smaller CNFs and faster unit propagation.

The runtime of the preprocessing is also plotted in relation to the time of solving the original CNF. For problems requiring between 30 seconds and 30 minutes to solve, preprocessing took less than 1/10th of the total time. Only for some of the easiest problems did preprocessing dominate runtime, but never in any really harmful way.

Study 3 – Runtime comparison solving with/without preprocessing. In *Figure 2* we plotted the preprocessing plus SAT solving time using the strongest version of our preprocessing (y-axis) against SAT solving without preprocessing

[6] www-cad.eecs.berkeley.edu/Respep/Research/vis/texas-97/

[7] www.sun.com/microelectronics/communitysource/picojava/download.html

[8] www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/BMC/bmc.tar.gz

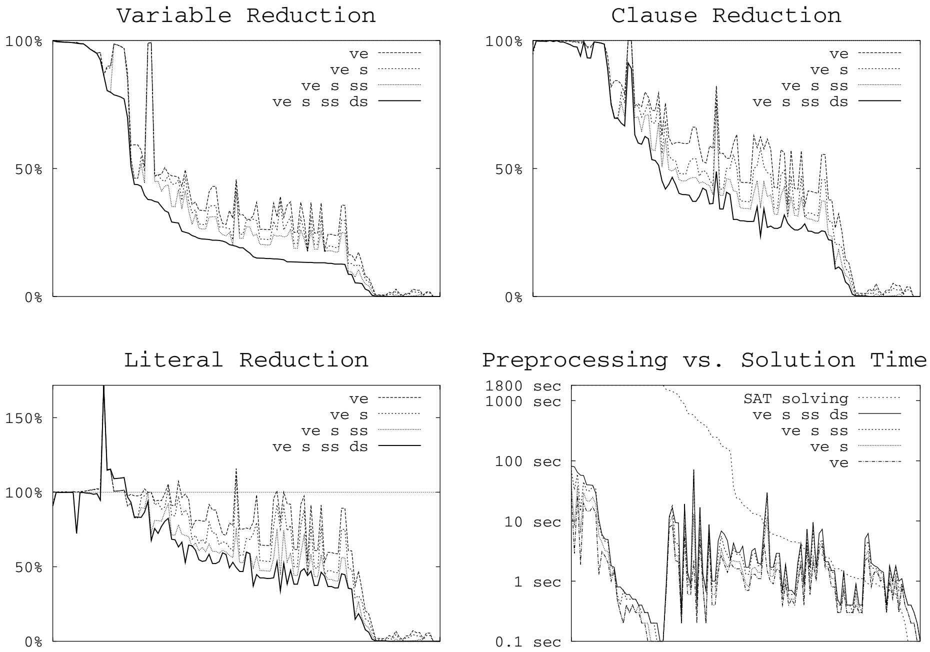


Fig. 1. *Relative Reduction and Preprocessing Time.* **The plots show the remaining size of *each one* of the 115 problems in the Industrial Mix** after reduction by our preprocessing techniques. The abbreviations are: "ve" variable elimination, "s" subsumption, "ss" self-subsumption, "ds" definitional substitution. The reduction is measured relative to the size of the original CNF after applying unit propagation (= 100%). The curves show the remaining variables (upper left), clauses (upper right), and literals (lower left). For all three plots the instances on the x-axis are sorted in the same way. **The order is determined by the percentage of remaining variables for the most *effective* version of the preprocessor** (the lower curve in the upper left plot labelled "ve s ss ds"). Our primary simplification target, the elimination of variables induces a simplification of the number of clauses in most cases as well. The number of literals follows more loosely the same trend. These three plots also show that our new simplification techniques are very effective compared to the approach taken by NiVER [1], which corresponds to the curves labelled "ve". Often an additional factor of two in reduction can be achieved.

The lower right plot shows in logarithmic scale the absolute time needed for preprocessing in relation to the overall solution time. The upper curve refers to the time for solving an instance with MiniSat not using preprocessing (timeout set to 1800 seconds). The remaining five curves show only the time used for preprocessing alone with decreasing effort. Preprocessing time turns out to be negligible compared to the overall solution time in most cases, even when our most aggressive techniques are used. Only for very simple instances is it better to run the solver without preprocessing

(x-axis). Although not a consistent improvement time-wise, in the big majority of cases preprocessing lead to a significant performance increase. In particular for ZChaff, the improvement was virtually exceptionless.

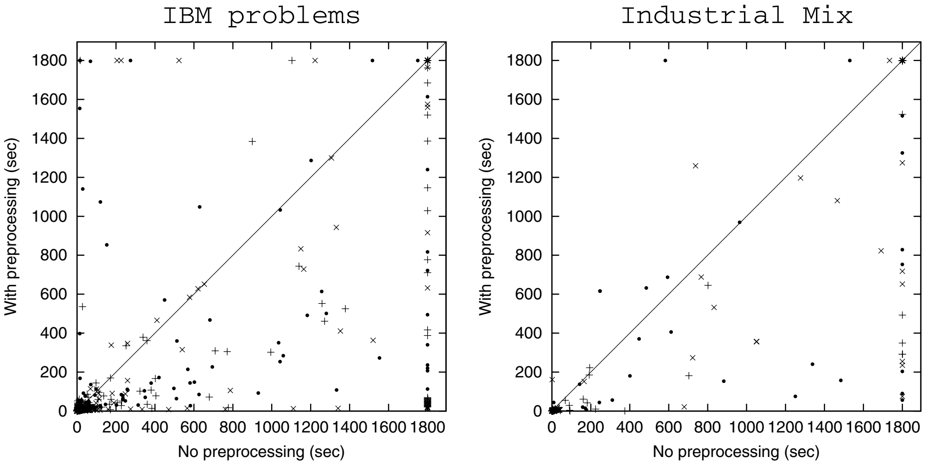


Fig. 2. *Heads-up comparison, with and without preprocessing.* The graphs shows a time comparison between "SAT solving" and "preprocessing + SAT solving" (using all reduction techniques). A mark below the diagonal means faster total solving time when first applying preprocessing. A dot (•) represents MiniSat, a plus (+) ZChaff, and a cross (×) BerkMin. A timeout of 1800 seconds was used, and marks along the edges represent tests which timed out for one of the two executions

Study 4 – Runtime effect of the different techniques. To evaluate the benefit of the different levels of reduction, we run all three SAT solvers on all benchmarks, both the IBM Problems and the Industrial Mix, with 5 different levels of optimization: (1) Nothing (original CNF after propagating unit clauses), (2) only variable elimination, (3) variable elimination plus subsumption, (4) variable elimination, subsumption and self-subsumption, (5) variable elimination using definitional substitution (when possible), subsumption and self-subsumption.

The result is plotted in *Figure 3.* The curves show that not only are more problems solved fast by preprocessing, but also more problems in total when a long timeout is given.

Study 5 – Incremental k-induction. In *Table 2* and *Figure 4*, a small study of applying our reduction techniques in an incremental context is presented. The internal SAT solver of SatELite, a less optimized version of MiniSat, allows SatELite to be used not only as a preprocessor, but also as an incremental SAT solver. Simplification is applied between each incremental SAT problem. Although this is a small study, the preliminary results suggest that our techniques pay off in an incremental context too.

6 Conclusion

New simplification techniques were presented together with important implementation details. On a large representative set of industrial benchmarks it was shown, that they speed up SAT solvers considerably. We also believe that pre-

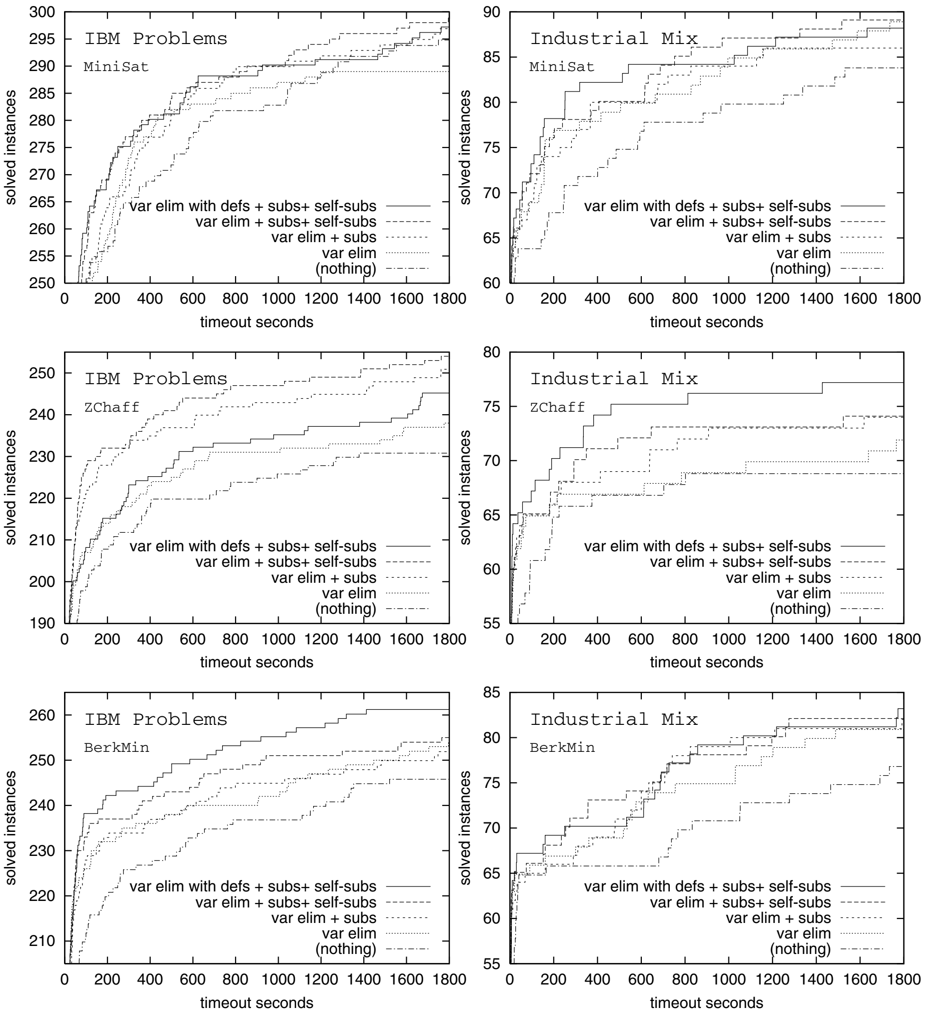


Fig. 3. *Comparing different preprocessing options using* SATELITE. Time includes both preprocessing and solving. Even though variable elimination by definitional substitution gives a consistent reduction compared to variable elimination by clause distribution, it is not a clear winner in terms of CPU time, but seems to depend on which solver you apply (the solid line vs. the long-dashed line). However, both lines are clearly above the "(nothing)"-line, representing no preprocessing. The addition of self-subsumption to normal subsumption seems to be a clear winner (often better, never worse). To get an estimate of the speedup, the graphs could be read by fixing a particular number of solved instances, and see what timeout is required to solve that number of instances. On the IBM benchmarks, MINISAT requires a timeout of about 250 seconds to solve 275 problems with full preprocessing, but a timeout of more than 600 seconds with no preprocessing

Table 2. *Study on k-induction.* We modified TIP [23] to use SATELITE as a back-end and ran the *zigzag* incremental induction algorithm on the "prodcell" problem distributed with VIS. The table shows the total runtime of each problem in seconds, omitting examples solved in less than 1 second. Within parenthesis, the number of clauses of the final incremental SAT instance is printed. In the rightmost column, all simplifications of SATELITE were invoked between each incremental step. The "depth" is the induction depth needed to prove the property (all properties are true)

Name	Depth	Plain		Simplifying	
vis.prodcell_12	29	62.7 s	(266k)	25.3 s	(88k)
vis.prodcell_14	16	7.8 s	(124k)	6.4 s	(29k)
vis.prodcell_15	23	30.5 s	(200k)	14.0 s	(56k)
vis.prodcell_17	27	64.5 s	(253k)	24.1 s	(76k)
vis.prodcell_18	13	7.5 s	(114k)	5.0 s	(35k)
vis.prodcell_19	22	19.2 s	(192k)	12.3 s	(55k)
vis.prodcell_23	13	9.5 s	(120k)	5.9 s	(37k)
vis.prodcell_24	37	120.5 s	(319k)	34.3 s	(94k)

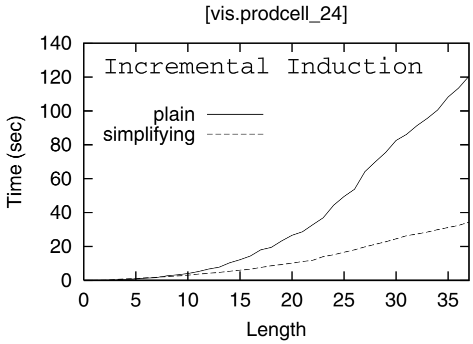


Fig. 4. *Case study on incremental k-induction.* The curve shows the progress of the temporal induction algorithm of TIP [23] running on the hardest example of *Table 2.* The graph plots the total execution time (y-axis) of all induction steps performed upto a certain length (x-axis). In this particular experiment, each step after length 25 takes more or less constant time

processing techniques partially provide a solution to the important problem of generating good CNFs in the application domain of circuit verification. As future work, it would be interesting to compare SAT solving time on problems that have been (1) clausified in a good way, and (2) clausified in a naive way, but processed with SATELITE. We also want to combine and compare our preprocessing techniques with the orthogonal techniques mentioned in the introduction.

Acknowledgements

Niklas Eén wants to thank Niklas Sörensson for setting him of in the direction of using self-subsumption in SAT.

References

1. Subbarayan, S., Pradhan, D.: NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. (In: Prel. Proc. SAT'04)
2. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Comm. of the ACM **5** (1962)
3. Bacchus, F., Winter, J.: Effective preprocessing with hyper-resolution and equality reduction. In: Proc. SAT'03. (Volume 2919 of LNCS.)
4. Brafman, R.: A simplifier for propositional formulas with many binary clauses. IEEE Trans. on Systems, Man, and Cybernetics **34** (2004)
5. Lynce, I., Marques-Silva, J.: Probing-based preprocessing techniques for propositional satisfiability. (In: Proc. ICTAI'03)
6. Novikov, Y.: Local search for boolean relations on the basis of unit propagation. (In: Proc. of DATE'03)
7. Stålmarck, G.: A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula (1989) Swedish Patent N° 467 076.
8. Kunz, W., Pradhan, D.: Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits. (In: Proc. ITC'92)
9. Kühlmann, A., Paruthi, V., Krohm, F., Ganai, M.: Robust boolean reasoning for equivalence checking and functional property verification. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **21** (2002)
10. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7** (1960)
11. Chatalic, P., Simon, L.: ZRes: The old Davis-Putnam procedure meets ZBDDs. In: Proc. CADE'00. (Number 1831 in LNAI)
12. Biere, A.: Resolve and expand. (In: Prel. Proc. SAT'04)
13. Boy de la Tour, T.: An optimality result for clause form translation. Journal of Symbolic Computation **14** (1992)
14. Jackson, P., Sheridan, D.: Clause form conversions for boolean circuits. (In: Prel. Proc. SAT'04)
15. Plaisted, D., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation **2** (1986)
16. Velev, M.: Efficient translation of boolean formulas to CNF in in formal verification of microprocessors. (In: Proc. ASP-DAC'04)
17. Tseitin, G.: On the complexity of derivation in propositional calculus. Studies in Constr. Math. and Math. Logic (1968)
18. Giunchiglia, E., Maratea, M., Tacchella, A.: Dependent and independent variables for propositional satisfiability. In: Proc. JELIA'02. (Volume 2424 of *LNCS.)*
19. Ostrowski, R., Grégoire, É., Mazure, B., Saïs, L.: Recovering and exploiting structural knowledge from CNF formulas. In: Proc. CP'02. (Volume 2470 of LNCS.)
20. Grégoire, É., Ostrowski, R., Mazure, B., Saïs, L.: Automatic extraction of functional dependencies. (In: Prel. Proc. SAT'04)
21. Eén, N., Sörensson, N.: An extensible SAT solver. In: Proc. SAT'03. (Volume 2919 of LNCS.)
22. Sörensson, N.: (Conflict clause simplification using subsumption resolution) paper in preparation.
23. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. In: Proc. BMC'03. (Volume 89(4) of ENTCS.)

Resolution and Pebbling Games

Nicola Galesi[⋆] and Neil Thapen[⋆⋆]

Abstract. We define a collection of Prover-Delayer games to characterise some subsystems of propositional resolution. We give some natural criteria for the games which guarantee lower bounds on the resolution width. By an adaptation of the size-width tradeoff for resolution of [10] this result also gives lower bounds on proof size.

We also use games to give upper bounds on proof size, and in particular describe a good strategy for the Prover in a certain game which yields a short refutation of the Linear Ordering principle.

Using previous ideas we devise a new algorithm to automatically generate resolution refutations. On bounded width formulas, our algorithm is as least as good as the width based algorithm of [10]. Moreover, it finds short proofs of the Linear Ordering principle when the variables respect a given order.

Finally we approach the question of proving that a formula F is hard to refute if and only if is "almost" satisfiable. We prove results in both directions when "almost satisfiable" means that it is hard to distuinguish F from a satisfiable formula using limited pebbling games.

1 Introduction

Propositional resolution is one of the most intensively studied logical systems. It is important both from applied and from theoretical points of view. On one hand it provides the logical basis for almost all of the more important and efficiently implemented automatic theorem provers (see [6]). On the other hand, it has probably been the most studied proof system in the area of proof complexity ([16, 7, 5, 10, 22] among others).

Most of the work to understand the strength of resolution has been concentrated on proving lower bounds for the length of refutations. Recently Ben-Sasson and Wigderson [10] based on ideas of [7] gave a unified approach to obtaining lower bounds. They showed that if a bounded width formula has a short refutation, then it has a narrow refutation. Using this relationship they give an algorithm to generate resolution refutations based on the width measure.

Simple pebbling games were initially introduced into the world of resolution in [20] to study size lower bounds in the subsystem of resolution where the proofs

[⋆] Dipartimento di Informatica. Università degli Studi di Roma "La Sapienza", Roma, Italy. Work done when the author was at Universitat Politècnica de Catalunya, Barcelona. Partly supported by Spanish grant TIC2001-1577-C03-02.

[⋆⋆] St Hilda's College. University of Oxford. Oxford, UK.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 76–90, 2005.

are treelike. Later the study of the space measure for resolution (see [13, 1, 8]) suggested the use of a more complex pebbling game.

It was soon clear that games also play an important role also in the study of the width limit for the full resolution system. Atserias and Dalmau [3] have given a finite model-theoretic characterization of the bounded width formulas with narrow refutations, using a pebbling game. Atserias, Kolaitis and Vardi in [4] gave a characterization, by combinatorial games from finite model theory, of refutational width in a refutational system tailored for constraint satisfaction problems.

In this paper we carry further the idea of using pebble games to study resolution. In section 3 we define a new modification of the resolution system (narrow resolution) for which we give a unified way of proving lower bounds similar to those for bounded width resolution of [10], but without having the restriction that the original formula has bounded width.

We define a "witnessing" pebble game, played between a Prover and a Delayer, and show that proofs in this system correspond to strategies for the Prover. On the other hand, a good strategy for the Delayer corresponds to the formula being extended dynamically satisfiable (EDS), an extension of an idea used in [12, 14] to prove lower bounds on space and on treelike size.

In section 4 we give some sufficient conditions for a formula to be k-EDS. In particular we show that if there is a satisfiable formula G which looks similar to F in that G cannot be distinguished from F with $k+1$ or fewer pebbles, then F is k-EDS. We also adapt a criterion of Riis' [21] to show that if F is a translation of a combinatorial principle on some finite structure, and this principle is satisfiable on a larger structure, then F is $\Omega(n)$-EDS. This gives us a useful sufficient condition for proving width lower bounds.

Starting from this witnessing pebble game we explore in two directions. In Section 5 we study some more general "structured" games, and generalize the width concept. In the earlier game the restriction on width corresponded to the Prover only being able to remember the values of a limited number of variables. In the structured game, the number of pebbles still limits how much the Prover can remember, so the games have useful properties in common with bounded width proofs; but each pebble can be labelled with information about several variables, which means the Prover can refute things that would be impossible if he was limited by width.

We show how to recover Resolution refutations from Prover strategies (theorem 20). In particular, since we show that the Prover has a winning strategy for a certain "ordered" structure game with only three pebbles over the Linear Ordering Principle LOP (theorem 26), we can recover a polynomial size refutation of this principle.

The LOP principle was used in [11] to prove the optimality of the width-size tradeoff for Resolution. It is one of the very few examples of CNF formulas having polynomial size resolution refutations but on which the Ben-Sasson Wigderson algorithm takes a very long (just subexponential) time to recover a refutation. Moreover it is also hard for DPLL-based theorem provers. Motivated by the previous result we then look at the the question of automatically generating resolution refutations using strategies for the game.

We devise a new algorithm based on reconstructing strategies for the Prover which allows us to extend the Ben-Sasson and Wigderson algorithm to an algorithm which, with some extra information about an ordering of the variables, generates short refutations of LOP in polynomial time. Moreover we show that for bounded width formula our algorithm is at least as good as the Ben-Sasson and Wigderson algorithm.

Finally in section 6 we explore further the idea used in section 4, that if a CNF F is similar (in some finite model theoretic sense) to a satisfiable formula G, then F can be thought of as "almost satisfiable" and is hard to refute. While this last result can be seen as a soundness theorem, in this section we also give a kind of "completeness" theorem, and show that if F is hard to refute using a certain game, then F is similar to a satisfiable CNF G.

Due to space limitations many proofs are shortened or omitted. A full version is available as an ECCC technical report [15].

2 Preliminary Definitions

Resolution is a refutation proof system for formulas in CNF form based on the following rule: $\frac{C\vee x \quad \bar{x}\vee D}{C\vee D}$ where if C and D have common literals, they appear only once in $C \vee D$. A resolution refutation of a CNF formula F is a derivation of the empty clause from the clauses defining F, using the above inference rule. As usual a refutation can be view as a directed acyclic graph.

The *size* of a refutation is the number of clauses in it. The *width* of clause is the number of literals in it; the width of a refutation is the maximal width of a clause in the proof; the width of refuting a formula is the minimal width of a refutation of that formula.

We consider two standard families of CNF, the *Pigeon Hole Principle*, PHP^m_n, with variables p_{ij} expressing "pigeon i goes to hole j":

$$\bigwedge_{i\in[m]} \bigvee_{j\in[n]} p_{ij} \quad \wedge \bigwedge_{i,i'\in[m],j\in[n],i\neq i'} (\neg p_{ij} \vee \neg p_{i'j})$$

and the linear ordering principle LOP_n, that expresses (the negation of) that every linear ordering of n elements has a least element, with variables $x_{i,j}$ expressing "i is less than j":

$$\bigwedge_{i,j,k\in[n]} (\neg x_{i,j} \vee \neg x_{j,k} \vee x_{i,k}) \wedge \bigwedge_{i,j\in[n]} (x_{i,j} \vee x_{j,i}) \wedge \\ \bigwedge_{i,j\in[n]} (\neg x_{i,j} \vee \neg x_{j,i}) \wedge \bigwedge_{i\in[n]} \bigvee_{j\in[n],i\neq j} x_{j,i}.$$

3 The Witnessing Game

Definition 1. *Fix $k \in \mathbb{N}$. Call a clause narrow if it has width k or less; otherwise it is wide. A width k* narrow resolution refutation *of a CNF F is a sequence* *of narrow clauses, beginning with the narrow clauses of F and finishing with the empty clause. There are three ways that clauses can be introduced into the sequence:*

1. *From B we can derive Bx (weakening);*
2. *From Bx and $C\bar{x}$ we can derive BC (resolution);*
3. *If $x_1 \dots x_m$ is a (usually wide) clause in F, then from $B\bar{x}_1, \dots, B\bar{x}_m$ we can derive B (resolution by cases).*

Lemma 2. *If F is a r-CNF with a width k narrow resolution refutation, then F has a width $r + k - 2$ "normal" resolution refutation.*

We introduce a pebble game to accompany this proof system.

Definition 3. *Let F be a CNF. The witnessing pebble game on F is played between a Prover and a Delayer on the set of literals arising from the variables in F. A pebble can never appear on both a literal and its negation. Normally we will limit the game to some number k of pebbles, and call this the k-pebble witnessing game. In each turn, one of three things can happen.*

1. *The Prover lifts a pebble from the board; the Delayer makes no response.*
2. *(Querying a variable) The Prover gives a pebble to the Delayer and names an empty variable x (that is, neither x nor $\bar{x}$ can be pebbled already). The Delayer must put the pebble on either x or $\bar{x}$.*
3. *(Querying a clause) The Prover gives a pebble to the Delayer and names a clause C from F. The Delayer must place the pebble on one of the literals in C, without contradicting any pebble already on the board. If this is not possible then the Prover wins.*

Notice that the Prover can win exactly when the pebbles on the board falsify some clause of F and the Prover has one pebble left over.

The next theorem describes the exact relationships between winning strategies for the Prover in the witnessing game, and refutational size and space.

Theorem 4. *Let F be a CNF and $k \in \mathbb{N}$.*

1. *If there is a winning strategy for the Prover in the k-pebble witnessing game for F, then there is a narrow resolution refutation of F of width k.*
2. *If there is a narrow resolution refutation of F of width k then there is a winning strategy for the Prover in the $(k+1)$-pebble witnessing game for F.*
3. *If F has a (normal) resolution proof of width k, then there is a winning strategy for the Prover in the $(k+1)$-pebble witnessing game for F.*
4. *If F has a (normal) resolution proof of clause space k, then there is a winning strategy for the Prover in the k-pebble witnessing game for F.*

Proof. We include a proof of 1, as it illustrates the correspondence between a strategy and a proof which we will be using throughout this paper. Consider the Prover's strategy as a tree, with each node labelled with the set of literals falsified under the assignment given by the pebbles currently in play. Then the root will be the empty clause, and the leaves will be (some subset of) the narrow clauses of F. If we read this tree from the leaves down to the root, we get precisely a narrow resolution refutation of F. Removing a pebble corresponds to weakening, the Prover querying a variable corresponds to a resolution step, and the Prover querying a clause corresponds to a resolution-by-cases step. □

We can now adapt the Ben-Sasson Wigderson result that "short proofs are narrow" to talk about games rather than proofs. This allows us to apply it directly to CNFs of unbounded width.

Definition 5. *If F is a CNF and x is a literal, we obtain $F|x$ from F by removing all clauses containing x and removing $\bar{x}$ from any clause in which it appears.*

So from a resolution refutation of F we can obtain a refutation of $F|x$ of equal size or smaller, by substituting in a value of "true" for x and simplifying.

Lemma 6. *If the Prover has a winning strategy for the k-pebble witnessing game on $F|x$, then in the k-pebble witnessing game on F the Prover can force the Delayer to either lose the game or place a pebble on $\bar{x}$.*

Proof. Let S be the k-pebble winning strategy for $F|x$. We will make this into a strategy for the game on F as follows. Whenever a clause C is queried in S such that $C \in F|x$ but $C \notin F$, it must be that $C\bar{x} \in F$. So replace this query with a query of $C\bar{x}$. Then either the Delayer must eventually place a pebble on $\bar{x}$, or the play of pebbles must be exactly the same as given in strategy S so that the Delayer must eventually lose. □

Theorem 7 ([10]). *Fix $d, n \in \mathbb{N}$ and let $\beta = (1 - \frac{d}{2n})^{-1}$. Say that a clause is fat if it has width greater than d. Then for any $m \leq n$ and any b, if F is a CNF on m variables and has a (normal) resolution refutation Π containing $< \beta^b$ many fat clauses, then the Prover has a winning strategy in the witnessing pebble game on F, using $d + b + 1$ pebbles. (See corollary 11 for an application of this.)*

Proof. The proof is by induction on m. The base case $m = 0$ is trivial, so suppose $m > 0$.

If $b = 0$, then every clause in Π (and also every clause in F) has width $\leq d$, so by an earlier observation there is a strategy for the Prover using $d+1$ pebbles.

Otherwise, let Π^* be the set of fat clauses appearing in Π. Then there must be some literal x appearing in at least $\frac{d}{2n}|\Pi^*|$ fat clauses, since otherwise $|\Pi^*|\, d \leq |\{(y, C) : \; y \text{ is a literal in } C \in \Pi^* \;\}| < 2m\frac{d}{2n}|\Pi^*|$.

The first part of the Prover's strategy is to force the Delayer to put a pebble on x. Now $F|\bar{x}$ contains only $m - 1$ variables and has a refutation with fewer than β^b fat clauses, so by the inductive hypothesis the Prover has a strategy for the game on $F|\bar{x}$ using $b + d + 1$ pebbles. Hence by the lemma the Prover can force the Delayer to satisfy x. Setting x to true will make all the clauses in Π

containing x vanish, so $F|x$ contains only $m-1$ variables and has a refutation with fewer than $(1-\frac{d}{2n})|\Pi^*| \leq \beta^{b-1}$ fat clauses. Hence the Prover has a winning strategy T for $F|x$ with only $b+d$ pebbles.

The Prover now leaves one pebble on x and uses the remaining $b+d$ pebbles to carry out strategy T on the remaining variables. As in the lemma, he must change T slightly to make it into a strategy for the game on F, by replacing queries to $C \in F|x \setminus F$ with queries to $C\bar{x}$. But the Delayer can never put a pebble on $\bar{x}$, because there is already a pebble on x. So the game plays just like the $F|x$ game with strategy T, and the Prover wins. □

4 Extended Dynamic Satisfiability

Along similar lines to those used in [3] for resolution width and in [12] for resolution space, we characterize the Delayer's strategy in the witnessing game, in terms of families of partial assignments for the formula. We also generalize this result by studying sufficient conditions that imply good strategies for the Delayer, along the lines of the approaches in [21] and [18] using first order model theory. The following definition was introduced in [12].

Definition 8. *A CNF F is k-dynamically satisfiable (k-DS) if there is a class R of partial assignments to the variables of F with the following properties:*

1. *R is closed under subset;*
2. *If $\alpha \in R$, $|\alpha| < k$ and C is any clause of F, then there is an extension $\beta \in R$ of α that satisfies C (in the sense that it makes at least one of the literals in C true, but does not necessarily assign a value to all the literals).*

To characterize good strategies for the delayer in our witnessing game, we alter the definition of dynamic satisfiability by adding a case to deal with queries made to variables:

Definition 9. *A CNF F is k extended-dynamically satisfiable (k-EDS) if there is a class R of partial assignments satisfying the conditions of definition 8, with the extra case:*

3. *If $\alpha \in R$, $|\alpha| < k$ and x is any variable appearing in F, then there is an extension $\beta \in R$ of α that assigns some value to the variable x.*

Lemma 10. *A CNF F is k-extended dynamically satisfiable if and only if the Delayer has a winning strategy for the k-pebble witnessing game on F.*

Proof. Suppose F is k-EDS. Then the Delayer can guarantee that after every turn the assignment α given by the k pebbles is in R. Hence by parts 2 and 3 of the definition of extended dynamic satisfiability, the Delayer is always able to consistently satisfy any variable or clause of F that the Prover queries. Conversely, suppose that the Delayer has a winning strategy. Let R be the set of all assignments corresponding to the configurations of pebbles that can appear in a game in which the Delayer uses this strategy. Then R witnesses that F is k-EDS. □

Corollary 11. *(to theorem 7) For any $\varepsilon > 0$, if a CNF F has n variables and is $(\sqrt{8}n^{\frac{1+\varepsilon}{2}} + 1)$-EDS, then it has no resolution refutation of size 2^{n^ε}.*

4.1 A Sufficient Condition for Extended Dynamic Satisfiability

We will treat CNFs as two sorted structures, with a clause sort and a variable sort and two binary relations "variable x appears positively in clause C" and "variable x appears negatively in clause C". We describe a kind of pebble game, the $(1, k)$-embedding game. The game is played between a Spoiler and a Duplicator on the set of clauses and variables of two CNFs F and G. Each player has k variable pebbles and one clause pebble.

Each turn the Spoiler either plays a clause pebble on one of the clauses of F, in which case the Duplicator must play his corresponding pebble on one of the clauses of G; or the Spoiler plays a variable pebble on one of the variables of either CNF, in which case the Duplicator must place his corresponding pebble on one of the variables of the other CNF.

At the end of each turn, the pebbles in play give a correspondence between some of the clauses and variables of F and those of G. The aim of the Duplicator is to make sure that this is a partial isomorphism. If at any point it is not, then the Spoiler has won.

Theorem 12. *Suppose G is satisfiable, and there is a winning strategy for the Duplicator in this game. Then there is a winning strategy for the Delayer in the k pebble witnessing game on F. Hence F is k-EDS.*

For an example, let F be PHP_n^{n+1} and G be PHP_n^n. Then G is satisfiable, and the Duplicator wins the $(1, n-2)$-embedding game on F and G (using a strategy generated by the set of partial injective mappings from the pigeons of F to the pigeons of G). This "almost satisfiability" of PHP seems to be related to the idea of critical assignments that is often used in hardness proofs for it.

In general this theorem is not easy to use, because it is not necessarily easy to prove that two CNFs F and G are in this relationship (although a sufficient condition is for F and G to be indistinguishable in the normal $k+1$ pebble game of finite model theory). We give an easier-to-use, but weaker, sufficient condition for extended dynamic satisfiability below, for the case where F and G are propositional translations of some first order principle. The argument is a standard one, see Krajicek [18, 17] or Riis [21] although we do not insist that the structure in which the principle is satisfied is infinite.

Theorem 13. *Let F be a CNF. Let ϕ be a first order quantifier free formula in a relational language L with equality. Let Φ be the formula $\forall x_1 \in [n_1] \ldots \forall x_k \in [n_k] \exists x_{k+1} \in [n_{k+1}] \ldots \exists x_l \in [n_l] \phi(\bar{x})$, where $n_1, \ldots, n_l \in \mathbb{N}$.*

Suppose that Φ is satisfiable "on a larger domain", that is, there is an interpretation in $\mathbb{N}$ of the relation symbols from L such that, with this interpretation, $\mathbb{N} \models \forall x_1 \in S_1 \ldots \forall x_k \in S_k \exists x_{k+1} \in S_{k+1} \ldots \exists x_l \in S_l \phi(\bar{x})$, where $S_1, \ldots, S_l$ are (possibly infinite) subsets of $\mathbb{N}$ with $|S_j| \geq n_j$ for each j and $n_i \geq n_j \rightarrow S_i \supseteq S_j$ for each i, j.

Let $\langle\Phi\rangle$ be the formula $\bigwedge_{i_1\in[n_1],\ldots,i_k\in[n_k]}\bigvee_{i_{k+1}\in[n_{k+1}],\ldots,i_l\in[n_l]}\phi(\bar{i})$. We treat this as a propositional formula in the usual fashion, thinking of atomic sentences involving a relation symbol as propositional variables, and of atomic sentences involving equality between numbers as the appropriate one of the connectives $\{T,F\}$. Let $n=\min\{n_1,\ldots,n_l\}$ and let r be the maximum arity of any relation symbol. Then $\langle\Phi\rangle$ is $(\frac{n}{r}-l+1)$-EDS (l is the number of variables).

It follows that if F is a CNF, and we can fix a one-to-one renaming of the propositional variables of $\langle\Phi\rangle$ with respect to which every clause of F is implied by some clause of $\langle\Phi\rangle$, and every variable in F appears in $\langle\Phi\rangle$, then F is $(\frac{n}{r}-l+1)$-EDS. (We call this condition "F is covered by Φ".)

Corollary 14. *LOP_n is $(\frac{n}{2}-3)$-EDS. (Note that LOP_n has n^2 variables, so corollary 11 does not apply.)*

Proof. LOP_n is covered by the formula $\forall x,y,z\in[n]\exists w\in[n],(\neg P(x,y)\vee\neg P(y,x))\wedge(P(x,y)\vee P(y,x))\wedge(P(x,y)\wedge P(y,z)\rightarrow P(x,z))\wedge P(w,x)$, and this is satisfiable if we let the quantifiers range over all of $\mathbb{N}$ and interpret P as any total ordering with no least element. □

5 Pebbling Games and Subsystems of Resolution

We represent (usually partial) assignments by the following notation: $[x_1\mapsto 1,\ldots,x_n\mapsto 0]$. Given an assignment α, we denote by $\overline{\alpha}$ the negation of α, that is, if α is the assignment $[x\mapsto 1,y\mapsto 0,z\mapsto 1]$ then $\overline{\alpha}$ is the clause $(\neg x\vee y\vee\neg z)$. Given a set S of assignments, we denote by $\overline{S}$ the set of clauses obtained by the negation of all assignments in S.

Definition 15 (Structure). *Let F be a CNF formula. For each clause C of F, let S_C be a set of partial assignments to the variables of C, such that*

1. *each assignment in S_C satifies C*
2. *C implies the disjunction of the assignments (in other words, $C\cup\overline{S_C}$ is a contradictory set of clauses).*

We call $\mathcal{S}=\bigcup_{C\in F}S_C$ a structure *for F.*

The *Structured Witnessing Game* $\mathcal{SWG}(F,\mathcal{S}_F)$, over a CNF F and with a structure $\mathcal{S}_F$, is a two player (Prover and Delayer) game defined as follows.

At each round the Prover either

- puts a pebble on some clause C of F. Then the Delayer answers by choosing one assignment $\alpha\in S_C$, and labelling the pebble with it; the Delayer is not allowed to choose an assignment inconsistent with an assignment already in play
- or removes a pebble, together with with its label.

The game ends when the Delayer is unable to choose an assignment consistently. This only happens when the the assignments labelling all the pebbles in play together falsify some initial clause of F.

We say that a formula F is $(k, \mathcal{S})$-easy if the Prover has a winning strategy for the game $\mathcal{SWG}(F, \mathcal{S})$ using at most k pebbles simultaneously. Otherwise the Delayer has a winning strategy, and we say that F is $(k, \mathcal{S})$-hard.

We will look at three structures in this paper.

- In the *unary* structure $\mathcal{U}_F$, for each $C \in F$, $S_C = \{[l \mapsto 1] : l \in C\}$. In other words, assignments in S_C satisfy exactly one literal in C. Note that the game for this structure is similar to the witnessing game described above, except that here the Prover is not allowed to query the value of individual variables.
- In the *ordered* structure $\mathcal{O}_F$, for each clause $C \in F$ we first fix a total order $\prec$ on the variables of C (extended to literals by ignoring negations), then define $S_C = \{(\bigcup_{r \in C, r \prec l}[r \mapsto 0]) \cup [l \mapsto 1] : l \in C\}$.
- In the *full structure* $\mathcal{F}_F$, for each clause C we let S_C be the set of all possible assignments to all of the variables in C.

In lemma 24 we show that the PHP^m_n is $(n/2 - 1, \mathcal{F})$-hard. On the other hand we also prove that LOP_n formulas are $(3, \mathcal{O})$-easy, and that this gives us an upper bound on the size of refutations of LOP_n.

Clearly the more complex the structure is, the more information the Prover can get using fewer pebbles, and the easier it will be to force the Delayer into a contradiction. This is captured by the following lemma.

Definition 16. *Let $\mathcal{S}_F$ and $\mathcal{T}_F$ be two structures for F. We write $\mathcal{S}_F \leq \mathcal{T}_F$ if for all $C \in F$, it holds that: (1) for all $\alpha \in S_C$ there is a $\beta \in T_C$ such that $\beta \subseteq \alpha$; (2) for all $\beta \in T_C$ there is $\alpha \in S_C$ such that $\beta \subseteq \alpha$.*

Lemma 17. *If F is $(k, \mathcal{S}_F)$-hard and $\mathcal{S}_F \leq \mathcal{T}_F$, then F is $(k, \mathcal{T}_F)$-hard.*

5.1 Feasible Structured Games and Short Refutations

Definition 18 (Feasible Structure). *Let F be a CNF formula. We say that a structure $\mathcal{S}_F = \bigcup_{C \in F} S_C$ is* feasible *if there are two polynomials p and q, such that for all $C \in F$:*

- *$|S_C| \leq p(|F|)$;*
- *there is a resolution refutation of $C \cup \overline{S}_C$ of size bounded by $q(|F|)$.*

Lemma 19. *The unary and the ordered structures are feasible.*

From a winning strategy for the Prover in a feasible structured game, we can construct a resolution refutation of F.

Theorem 20. *Let F be CNF with m clauses and let $\mathcal{S}_F$ be a feasible structure for F. If F is $(k, \mathcal{S}_F)$-easy, then there is a resolution refutation of F of size bounded by $O(m^k|\mathcal{S}_F|^k)q(|F|)$.*

Proof. (sketch) Think of the Prover's strategy as a dag. Each node is labelled with a position in the game and the query that the Prover makes from that position, hence there are at most $O(m^k|\mathcal{S}|^k)$ nodes. There is an edge going out from a node for each possible reply that the Delayer makes.

We make this strategy into a resolution refutation as in the proof of theorem 4. We negate the assignments at each node so that the nodes are now labelled with clauses, and we reverse the direction of all the edges. Finally we replace all the edges now coming into a node with the resolution refutation (of $C \cup \bar{S}_C$ from the definition of feasible structure) corresponding to the clause C being queried at that node. □

5.2 Structured Games as Subsystems of Resolution

In [14] it is proved that k-dynamic satisfiability is a sufficient condition for a CNF F to require exponential size treelike resolution refutations. In the next theorem we prove that it completely characterizes Delayer strategies for the unary structure.

Theorem 21. *F is k-dynamically satisfiable if and only if it is $(k,\mathcal{U})$-hard.*

Extended dynamic satisfiability corresponds to the witnessing game, in which the Prover is allowed to query variables. That is not allowed in our structured games, but we do have the following relationship.

Theorem 22. *If F is $(k,\mathcal{F})$-hard (that is, hard for the full game) then F is k-EDS.*

Now we will consider strategies for the Prover and Delayer for two standard families of CNFs, the pigeonhole principle PHP^m_n and LOP_n:

From definition 16 and lemma 17, it is straighforward to observe that:

Lemma 23. *$\mathcal{F} \leq \mathcal{O} \leq \mathcal{U}$.*

It is quite easy to prove that the PHP^m_n is hard for all these games:

Lemma 24. *PHP^m_n is $(n/2-1,\mathcal{F})$-hard.*

On the other hand, using the fact that LOP_n is $(\frac{n}{2}-3)$-dynamically satisfiable (see [12] or corollary 14), we have:

Corollary 25. *LOP_n is $(\frac{n}{2}-3,\mathcal{U})$-hard.*

The main result of this subsection follows from the next theorem.

Theorem 26. *LOP_n is $(3,\mathcal{O})$-easy.*

Proof. Consider the following order on all variables, that in particular defines an order on each clause: $x_{i,j} \prec x_{h,k}$ iff either $j < k$ or $j = k$ and $i < h$.

We describe the strategy of the Prover by stages: we prove that at each stage the Prover, using only three pebbles, either wins or will force the Delayer to answer to a clause of the form $\bigvee_{j\in[n],j\neq r} x_{j,r}$ with an assignment assigning strictly more literals to 0 than the previous stage. Hence, if he does not win sooner, after at most n stages he will force a clause of this form to be falsified.

Assume w.l.o.g. that at the begining of a stage the Prover pebbles the clause $C_1 = \bigvee_{j\in[n],j\neq 1} x_{j,1}$. Let $\alpha \in S_{C_1}$ be the assignment chosen by the Delayer. α will be of the form $[x_{2,1} \mapsto 0, \ldots, x_{j-1,1} \mapsto 0, x_{j,1} \mapsto 1]$ for some $j \in [n], j \neq 1$. The Prover then pebbles the clause $C_j = \bigvee_{k\in[n],k\neq j} x_{k,j}$ Let β be the assignment chosen by the Delayer. β is of the form $[x_{1,j} \mapsto 0, \ldots, x_{k-1,j} \mapsto 0, x_{k,j} \mapsto 1]$ for some $k \in [n], k \neq j$.

Now if $k < j$, then $\alpha(x_{k,1}) = 0$. But then all literals in the clause $(\neg x_{k,j} \vee \neg x_{j,1} \vee x_{k,1})$ are false, and the Prover can pebble this clause and win.

Assume then that $k > j$. The Prover then pebbles the clause $\neg x_{k,j} \vee \neg x_{j,k}$. Since $\beta(x_{k,j}) = 1$, The Delayer must answer with the assignment γ setting $x_{j,k}$ to 0 (and obviously $x_{k,j}$ to 1). At this point the Prover removes the pebbles from C_1 and C_j and places a pebble on C_k. The Delayer must answer with an assignment δ of the form $[x_{1,k} \mapsto 0, \ldots, x_{l-1,k} \mapsto 0, x_{l,k} \mapsto 1]$ where clearly $l > j$, to not contradict the assignment γ. □

5.3 Automatic Generation of Refutations

These results show that the *ordered structure*, via theorem 20 and lemma 19, gives rise to a subsystem of daglike resolution (corresponding to a strategy for the Prover with $O(1)$ pebbles): (1) powerful enough to obtain polynomial size refutations of important families of contradictions like LOP_n; and (2) where PHP (and other classes of formulas we omit in this version) are hard to refute and the hardness proof is relatively easy.

Since LOP_n is an example of formulas known to be hard for many automatic theorem provers (see [6]), the previous properties suggest that it is worth investigating algorithms for generating winning strategies for the Prover, as this gives a way of generating resolution refutations.

We present such an algorithm below. It is analagous to the Ben-Sasson Wigderson algorithm based on width. In our case, rather than limiting the width, we limit the number of pebbles used in the strategy.

We first describe a subroutine. The input is a formula F, and the description of the structure $\mathcal{S}_F$ (this may be given in a simple way, e.g. as an ordering if we are dealing with the ordered game), and a number k of pebbles. The output is a winning strategy for the Prover using k pebbles, if one exists, otherwise "NONE".

The subroutine first builds up a dag A as follows: The nodes of A are all of the positions possible in the game, ie. all the possible ways of labelling k or fewer pebbles plus a source node. There are $\leq O(m^k|\mathcal{S}_F|^k)$ of them, where m is the number of clauses in F. At the start of the algorithm all the positions that are self contradictory or falsify clauses of F are *marked*, and the graph has no edges. While the source node has not been marked the algorithm does the following: it checks each not marked node X in A. If by removing a pebble the Prover can move from X to a node Y already marked in A, then the algorithm marks X

and labels it with the pebble to be removed and adds an edge from X to Y. If there is a clause C that is not pebbled at X, and is such that whatever label the Delayer could choose to give to C, it would lead to a position corresponding to a node already marked in A, then the algorithm marks X, labels it with C, and adds edges going to all the nodes corresponding to the Delayer's possible answers.

The subroutine stops when the source node has been marked, or when there are no more edges to be added to the graph. If the source node has been marked, the subroutine outputs only the marked subgraph of A, which is a winning strategy for the Prover. Otherwise it outputs NONE.

The proof search algorithm works by calling this subroutine for increasing values of k, until the subroutine outputs a strategy (which it will do eventually, when the Prover is able to query all clauses at once).

The running time of the algorithm is $O(m^{r+1}|\mathcal{S}_F|^{r+1})$ where r is the minimal number of pebbles used by the Prover to win $\mathcal{SWG}(F, \mathcal{S}_\mathcal{F})$.

For feasible structures, using theorem 20, this algorithm allows us to generate daglike resolution refutations of size polynomial in the size of the formula.

For the case of ordered structures we could add a preprocessing phase to try all possible orders. Although in the worst case this can increase the running time to exponential we notice that for LOP_n, a good ordering is the one described in theorem 26, and this (or one equally good) arises very naturally from the first order combinatorial principle corresponding to the LOP_n formula, together with *any* ordering of the numbers $1, \ldots, n$.

Moroever we observe also that although the full structure is not feasible in general, it become so in the case of formulas with $O(1)$ initial width. Now by theorem 4, lemma 10 and theorem 22, if F has width k refutations in resolution then F is $(k+1, \mathcal{F})$-easy, so if there is a constant width formula with a narrow refutation then our algorithm (looking for strategies for the full game) works at least as well on it as the Ben-Sasson and Wigderson algorithm.

6 Satisfiability vs Hardness

In Section 4 we showed how the existence of a satisfiable formula G similar to F gives a good strategy for the Delayer in a certain game, which shows that F has no narrow resolution refutation.

This suggests an attractive idea, that we should look for a sort of soundness and completeness theorem for polynomial size resolution. It would have the following form: a CNF F has no small resolution refutation if and only if F is "almost" satisfiable, in that it is hard to distinguish from a satisfiable formula G.

In this section we give two results in this direction. We first show that if F is hard to distinguish from G in the sense that it is hard to prove in resolution that F and G are different, then F is hard to refute. We then show a converse, although this talks about games rather than proofs: if there is a good strategy for the Delayer in the full (structured) pebble game on F, then there exists a satisfiable CNF G that looks similar to F, in a certain sense.

Definition 27. *Let F and G be CNFs, considered as two-sorted structures. We define a new CNF, $ISO(F, G)$, that expresses the statement "F is isomorphic to G". This is intended to be used when F is not isomorphic to G, as a generalization of the pigeonhole principle. Let $Var(F)$ and $Cl(F)$ be the sets of variables and clauses in a CNF.*

We take the conjunction of

1. *$\bigvee_{D \in Cl(G)} \sigma_{CD}$ for each $C \in Cl(F)$; $\bigvee_{C \in Cl(F)} \sigma_{CD}$ for each $D \in Cl(G)$;*
2. *$\neg\sigma_{CD} \vee \neg\sigma_{CD'}$ and $\neg\sigma_{CD} \vee \neg\sigma_{C'D}$ for all $C \neq C' \in Cl(F)$, $D \neq D' \in Cl(G)$;*
3. *$\bigvee_{y \in Var(G)} \sigma_{xy}$ for each $x \in Var(F)$; $\bigvee_{x \in Var(F)} \sigma_{xy}$ for each $y \in Var(G)$;*
4. *$\neg\sigma_{xy} \vee \neg\sigma_{xy'}$ and $\neg\sigma_{xy} \vee \neg\sigma x'y$ for all $x \neq x' \in Var(F)$, $y \neq y' \in Var(G)$;*
5. *If x appears positively in C in F, but y does not appear positively in D in G, then we include the clause $\neg\sigma_{xy} \vee \neg\sigma_{CD}$; similarly for appearing negatively or not appearing at all.*

So 1 and 2 say that σ is a bijection on clauses, 3 and 4 say it is a bijection on variables, and 5 says it preserves the structure.

Theorem 28. *Suppose that G is satisfiable and F has a small resolution refutation. Then $ISO(F, G)$ has a small resolution refutation.*

Proof. (sketch) Let α be a satisfying assignment to G. α partitions $Var(G)$ into a set A of true variables and a set B of false variables. For $x \in Var(F)$, let X be the clause $\bigvee_{y \in A} \sigma_{xy}$ and let $\bar{X}$ be the clause $\bigvee_{z \in B} \sigma_{xz}$. Let F^* be F with every variable x replaced by X and every negated variable $\neg x$ replaced by $\bar{X}$.

We can derive F^* from $ISO(F, G)$, and then use the small refutation of F to give a small refutation of F^*. □

To use this to get lower bounds for F, we need lower bounds for $ISO(F, G)$. This is a generalization of the pigeonhole principle, so the many existing tools for showing lower bounds for PHP may be useful here. The most tractable case should be when F and G both arise as translations of some first order combinatorial principle, but on structures of slightly different sizes, so we can use the hardness of PHP directly. Krajicek gives an argument like this is in [18], relating the proof complexity of the weak pigeonhole principle and the Ramsey theorem. Potentially this will give a sufficient criterion for a formula to have no small daglike refutation, similar to Riis' criterion for treelike refutations [21].

So far we have given "soundness" theorems, that if F is almost a satisfiable CNF then F is hard to refute (or that it is hard for the Prover to win some game). Now we give a "completeness" theorem.

We think of CNFs as two sorted structures with a clause sort and a variable sort. There are two binary relations, "x appears positively in C" and "x appears negatively in C."

We define the sense in which our constructed CNF G will be similar to F. Informally, we have "local" embeddings of the clauses of F into the clauses of G. (If we had a global embedding, then the satisfying assignment for G would immediately give a satisfying assignment for F.)

Definition 29. *Let F and G be CNFs. We say that F is strongly k-clause embeddable in G if there is a family H of partial isomorphisms from F to G with the following properties:*

1. *Suppose $f \in H$ maps clauses $C_1, \ldots, C_k$ and no other clauses. Then the variables mapped by f are precisely the variables appearing in $C_1, \ldots, C_k$. In other words, f is first of all a partial isomorphism on clauses, but brings with it a partial isomorphism on the variables appearing in those clauses.*
2. *If $f \in H$ maps fewer than k clauses, and C is any other clause in F, then there is $g \in H$ that extends f and maps C somewhere.*

Lemma 30. *If F is strongly k-clause embeddable in G and G is satisfiable, then the Delayer wins the k-pebble full game on F.*

We could apply the following theorem to PHP_n^{n+1} (with $k = n/2 - 1$). Unfortunately the formula G that it gives does not seem to be as elegant as PHP_n^n.

Theorem 31. *Suppose the Delayer wins the k-pebble full game on F. Then there is a satisfiable finite CNF G such that F is strongly k-clause embeddable in G.*

Proof. We will build G out of the Delayer's strategy for the game. Think of this strategy as a dag consisting of positions in the game. Each position P is a tuple $(C_1, \alpha_1), \ldots, (C_r, \alpha_r)$ of at most k clauses from F together with assignments to all the variables appearing in each clause, that are consistent and satisfy each clause. Notice that only a finite number of different positions appear in the strategy.

Let G' be the CNF whose clauses are all the pairs (C, α) that appear in the Delayer's strategy. We give variables to the clauses as follows: if $C \in F$ contains variables $x_1, \ldots, x_m$, then (C, α) contains variables $(x_1, C, \alpha), \ldots, (x_m, C, \alpha)$ and (x_i, C, α) appears positively or negatively in (C, α) just as x_i appears positively or negatively in C. That is, G' consists of clauses named by pairs (C, α) and variables named by triples (x, C, α); each clause is isomorphic to some clause from F, and no two clauses share any variables. Now define an equivalence relation $\sim$ on the variables of G' as the transitive closure of the relation: (x, C, α) is related to (y, D, β) if and only if x and y are the same variable (in F) and (C, α) and (D, β) appear together in some position in the Delayer's strategy.

Let G be the CNF obtained from G' by renaming every variable (x, C, α) with its $\sim$-equivalence class $[(x, C, \alpha)]$.

The proof is completed by showing that G is satisfiable, and that F is strongly k-clause embeddable in G. □

References

1. M. Alekhnovich, E. Ben-Sasson, A. Razborov, A. Wigderson. Space complexity in propositional calculus. *SIAM J. Comput.* 31(4) pp. 1184–1211, 2002.
2. M. Alekhnovich, A.A. Razborov. Resolution is not automatizable unless $W[P]$ is not tractable. *42nd IEEE Symposium on Foundations of Computer Science, FOCS 2001*, pp. 210–219.

3. A. Atserias, V. Dalmau. A combinatorial characterization of Resolution Width *18th IEEE Conference on Computational Complexity* (CCC), pp. 239-247, 2003.
4. A. Atserias, P. Kolaitis, M. Vardi. Constraint Propagation as a Proof System. In *10th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS vol. 3258, pp. 77-91, 2004.
5. P. Beame, R.M. Karp, T. Pitassi, M.E. Saks. On the Complexity of Unsatisfiability Proofs for Random k-CNF Formulas. *SIAM J. Comput.* 31(4) pp. 1048–1075, 2002.
6. P. Beame, H. Kautz. A Sabharwal Understanding the power of clause learning *Proceedings IJCAI* pp. 1194–1201, 2003
7. P. Beame, T. Pitassi. Simplified and Improved Resolution Lower Bounds. *37th IEEE Symposium on Foundations of Computer Science, FOCS 1996*, pp. 274–282.
8. E. Ben-Sasson, N. Galesi. Space Complexity of Random Formulae in Resolution. *16th IEEE Annual Conference on Computational Complexity, CCC 2001*, pp. 42–51.
9. E. Ben-Sasson, R. Impagliazzo, A. Wigderson. Near optimal separation of tree-like and general Resolution. *Electronic Colloquium on Computational Complexity (ECCC) TR00-005, 2000.* To appear in *Combinatorica*.
10. E. Ben-Sasson, A. Wigderson. Short Proofs Are Narrow—Resolution Made Simple. *J. ACM* 48(2) pp. 149–168, 2001.
11. M.L. Bonet, N. Galesi. Optimality of Size-Width Tradeoffs for Resolution. *Computational Complexity*, Vol 10(4) 2001. pp. 261-276.
12. J.L. Esteban, N. Galesi, J. Messner. On the Complexity of Resolution with Bounded Conjunctions. *Theoretical Computer Science* 321(2-3) pp. 347–370, 2004.
13. J.L. Esteban, J. Torán. Space bounds for Resolution. *Inform. and Comput.* 171 (1) pp. 84–97, 2001.
14. N. Galesi, N. Thapen. The Complexity of Treelike Systems over λ Local Formuale *Proceedings of IEEE Conference on Computational Complexity* 2004.
15. N. Galesi, N. Thapen. Resolution and Pebbling Games *ECCC Technical Report TR04-112. http://www.eccc.uni-trier.de/eccc-reports/2004/TR04-112/index.html*
16. A. Haken. The Intractability of Resolution. *Theoret. Comp. Sci.* 39, pp. 297–308, 1985.
17. J. Krajíček. Bounded arithmetic, propositional logic, and complexity theory, Encyclopedia of Mathematics and Its Applications, Vol. **60**, *Cambridge University Press*,(1995),
18. J. Krajíček. On the weak pigeonhole principle. *Fund. Math.* 170(1-3) pp. 123–140, 2001.
    J. Symbolic Logic 59(1) pp. 73–86, 1994.
19. P. Pudlak. Proofs as Games. *American Math. Monthly*, Vol. 2000-2001, pp.541-550
20. P. Pudlák, R. Impagliazzo. A lower bound for DLL algorithms for k-SAT". *Conference Proceeding of Symposium on Distributed Algorithms* (2000), pp. 128-136.
21. S. Riis. A complexity gap for tree-resolution. *Computational Complexity* 10(3), pp. 179-209, 2001.
22. A. Urquhart. Hard examples for Resolution. *J. ACM* 34(1) pp. 209-219, 1987.

Local and Global Complete Solution Learning Methods for QBF

Ian P. Gent[1] and Andrew G.D. Rowley[2]

[1] School of Computer Science, University of St. Andrews, St. Andrews, Fife, UK
ipg@dcs.st-and.ac.uk

[2] Manchester Computing, Kilburn Building, Oxford Road, Manchester, UK
andrew.rowley@manchester.ac.uk

Abstract. Solvers for Quantified Boolean Formulae (QBF) use many analogues of technique from SAT. A significant amount of work has gone into extending conflict based techniques such as conflict learning to solution learning, which is irrelevant in SAT but can play a large role in success in QBF. Unfortunately, solution learning techniques have not been highly successful to date. We argue that one reason for this is that solution learning techniques have been ‘incomplete’. That is, not all the information implied in a solution is learnt. We introduce two new techniques for learning as much as possible from solutions, and we call them complete methods. The two methods contrast in how long they keep information. One, Complete Local Solution Learning, discards solutions on backtracking past a previous existential variable. The other, Complete Global Solution Learning, keeps solutions indefinitely. Our detailed experimental analysis suggests that both can improve search over standard solution learning, while the local method seems to offer a more suitable tradeoff than global learning.

1 Introduction

Quantified Boolean Formulae (QBFs) can be seen as an extension of Satisfiability (SAT) with the addition of a prefix in which every variable is universally or existentially quantified. This increase in expressivity comes at a price; the decision problem for QBF is PSPACE-complete. This allows QBF to be used to solve many other PSPACE problems such as games, conditional planning problems and hardware verification.

There have been many advances in QBF solver technology. Many of the more recent algorithms are based on an extension to the DPLL [3] procedure for SAT. The first such example is presented in [2]. Many other SAT techniques have been extended and added to this procedure, starting with conflict and solution directed backjumping [6], in which pruning of the search tree is achieved through look-back techniques. This provided a great improvement in both run-time and branching rate. Learning was then introduced [5, 7, 10]. This had much less impact than occurred in SAT. While improvements were seen on some problems, the algorithms actually performed worse on others.

Recently, it has been shown that solution learning for quantified Boolean formulae is not very good in practice. In two separate papers [5, 10], the experimental analysis of solution and conflict learning show that their combination does not perform well in practice. Two other papers [7, 9] show that conflict learning for QBF used independently

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 91–106, 2005.

performs quite well. This implies that the detrimental effect of the combination learning solver was likely to be due to the solution learning part of the solver. This effect is bad enough to make the combination of conflict and solution learning perform poorly.

There are at least two reasons why solution learning might not perform well in practice. Firstly, the level of solution learning performed was unrestricted learning. This means that when a solution is learned, it is kept for the remainder of the search. An alternative approach would be to perform some sort of bounded learning. This was suggested in [5] but was never implemented. A second reason is that the solutions learned may not be the most useful in practice. As was stated in [10], there are many possible sets which can be learned on discovery of a solution. There may be an exponential number of such solutions which would result in an exponential space requirement if all solutions are learned using this technique.

We introduce a method of solution learning which addresses both of the aforementioned issues. This technique will allow the learning of all solutions without the exponential space requirement. Furthermore, this technique will automatically throw away learned information when it is no longer of any use. This will be a requirement of the information as it will become invalid under certain conditions. A second technique will also be introduced which, in extension to the first technique, will allow the storage of the learned information for the duration of the search. This will allow other bounding algorithms to be applied to the learning.

First, a proper introduction to solution learning in QBF will be given. This will be followed by a description of the first new technique. The extension to this technique will then be described. Finally, an experimental evaluation of the techniques will be given. This will show that the new technique performs better than the standard solution learning technique on many instances.

2 Background

We provide some basic details of QBF, but refer the reader elsewhere for a more detailed introduction [2]. A QBF is of the form QC where Q is a sequence of quantifiers $Q_1v_1...Q_nv_n$, where each Q_i quantifies ($\exists$ or $\forall$) a variable v_i and each variable occurs exactly once in the sequence Q. C is a Boolean formula in conjunctive normal form (CNF), a conjunction of clauses where each clause is a disjunction of literals. Each literal is a variable and a sign. The literal is said to be negative if negated and positive otherwise. Universals and existentials are those variables quantified by a universal or existential quantifier respectively. A QBF of the form $\exists v_1 Q_2v_2...Q_nv_nC$ is true if either $Q_2v_2...Q_nv_nC[v_1 := T]$ or $Q_2v_2...Q_nv_nC[v_1 := F]$ is true, where T represents true and F represents false. Similarly, a QBF of the form $\forall v_1 Q_2v_2...Q_nv_nC$ is true if both $Q_2v_2...Q_nv_nC[v_1 := T]$ and $Q_2v_2...Q_nv_nC[v_1 := F]$ are true.

A QBF is trivially *false* if it contains an all-universal clause or an empty clause. An all-universal clause is a clause in which all the literals are universal. Such a clause is false since we must be able to satisfy the formula for all valuations of universals, including the combination which makes all literals false in the all universal clause. This is true so long as the clause is not a tautology, which we remove with preprocessing. A QBF is trivially *true* if it consists of an empty set of clauses.

The basic procedure for solving QBF's, [2], is similar to the DPLL procedure for SAT [3]. The order of variables is now determined by the order in which the variables occur in the prefix. Secondly, when the formula is found true, the procedure backtracks to the most recently assigned universal that has not had both true and false values explored. Further propagation is possible using unit or single existential clauses. A unit or single-existential clause is defined to be a clause in which only one existential variable is left to be assigned and all universal variables are quantified after the existential in the quantifier prefix. In this clause, the existential must be assigned true to avoid an all-universal clause.

Conflict Directed Backjumping[8] is not covered in this paper. For a detailed description of Conflict Directed Backjumping for QBF see [6]. **Solution Directed Backjumping** can be performed during QBF search when assignments result in an empty set of clauses. The first stage is to find a small set of universal literals from the current assignment which leaves the formula satisfied. This is known as a reason for the solution or the solution set. To find the solution set, each clause is examined in turn. Where a clause is satisfied by at least one existential, nothing is added to the solution set. Where a clause is only satisfied by a universal assignment, we choose one of the universals in the clause to be added to the set. We only backtrack on a universal variable when it occurs in the solution set. When both true and false assignments to a universal result in the QBF formula being true, we can combine the solution sets from both these assignments and pass the new solution set back to be used for universals assigned earlier. For more details see [6].

The idea behind **solution learning** is simple. The solution sets which are generated by solution directed backjumping are stored at certain points in the search. These then guide search away from previously visited solutions and thus reduce the amount of search performed. The solution set cannot be learned as it is. Solution sets were introduced as only containing universals. These universals come from the clauses that are not satisfied by any existentials. For solution learning, the solution set must be augmented with a subset of the existentials that satisfy the remaining clauses. This is because the solution is not caused only by the universals. Only universals are added to the solution set in solution directed backjumping because solution backtracking is only performed on universals.

In [5], solution learning was first introduced. It was realised here that not all solution sets could be learned as they were. The criterion for learning a solution set is that the solution set must be prefix closed. This means that for any variable v that occurs in the solution set, every variable quantified before v in the prefix must also occur. This will not always be the case in the solution set and so not all solution sets are learnable.

In [10], solution learning is performed using consensus for DNF cubes (the dual of resolution of CNF clauses). The QBF CNF formula C is extended to initially contain a DNF formula D such that $C \iff C \vee D$. Initially D is empty and is thus false. When a new solution is learned, it is added to D as a DNF cube. In conflict learning by resolution for SAT, resolution is performed on the clause that became empty, e, to cause the conflict and the clause that was a unit clause immediately before the last variable was assigned, u. The result of this is then resolved with the clause that became unit before u. This continues until an assignment is encountered that was not performed by propagation. At this point, the resolvent is added as a clause. Solution learning proceeds in the same way, except that the initial cube does not exist. This is instead created from a solution set. Where solution learning is referred to in the remainder of this paper, it is this method that will be considered.

We contrast '**local solution learning**' with existing QBF solution learning techniques, which we call **global solution learning**. In global solution learning [5, 10], the solution set is augmented with existentials. Otherwise, the solution set would only be valid under certain conditions. Hence a DNF cube generated from the solution set would have to be removed if these conditions change. In local solution learning, the learned cube must be removed when one of the existential variables is changed that is quantified outside one of the universal variables in the cube. This ensures that local learning only holds valid learned cubes. This is the 'local' part of the learning; the learned information is only valid in a local sub-tree in the search tree. Unless a learned cube removal system (such as relevance or size learning) is used, global learning will keep all learned cubes until the end of the search. This can result in an exponential increase in the size of the formula. Local solution learning can be considered a learned cube removal system. The size of the learned cubes will also be smaller in local solution learning, since they do not contain any existentials.

3 Complete Local Solution Learning

We call existing solution learning techniques '**incomplete**'. This indicates the fact that they do not learn all the information possible from each solution found. The methods we introduce are '**complete**', in that any solution is reused directly whenever possible, avoiding as much search as possible within the scope that solutions are preserved. The two methods differ in the scope within which found solutions are preserved, one being local and the other global. We introduce the local method first as the global method is based on it but is more complicated.[1]

QBFs of the form $\forall\exists B$. For simplicity, we first discuss the complete local solution learning for formulae which have a prefix of the form $\forall\exists$.

We use an example to show what incomplete learning misses. Consider the formula $\forall u_1 \forall u_2 \ldots \forall u_n \exists e_1 \ldots \exists e_m [(u_1 \vee u_2 \vee e_1) \wedge (u_3 \vee u_4 \vee e_2) \wedge \ldots]$. Say the formula is satisfied by the assignments where $e_1 = e_2 = F$, so that the two clauses are only satisfied by universals. Let the assignments to all the existentials in the current assignment be called E_1. Say now that the universal variables are all assigned true in the order u_1, u_2, u_3, u_4. So there are four possible solution sets that can be built. These are $\{u_1, u_3\}$, $\{u_1, u_4\}$, $\{u_2, u_3\}$ and $\{u_2, u_4\}$. Incomplete solution learning will only learn one of these sets. Yet, if we ever find ourselves assigning the universals true from *any* of the four sets, E_1 can still be used to satisfy the remaining clauses. Incomplete learning methods cannot exploit this fact, leading to unnecessary backtracks. Redundant search could be avoided in solution learning if all of the possible solution sets were learned at this stage, but this has the potential for causing an exponential increase in the size of the formula. Suppose

[1] Complete Local Solution Learning is the same algorithm as GR-Learning reported in our technical report [4], but we hope with a more descriptive name. It has been reimplemented, however, so results in this paper are new.

instead that the assignments to the universals were $u_1 = T, u_2 = T, u_3 = T, u_4 = F$. There are now two solution sets available: $\{u_1, u_3\}$ and $\{u_2, u_3\}$. Now when backjumping assigns $u_3 = F$, the algorithm will be free to choose $u_4 = T$ to satisfy the second clause. Again the original existential assignment E_1 can be used to satisfy the remaining clauses. Even if existing solution learning methods were to learn all the solution sets, this situation would not be covered. This is because solution learning is restricted to the set of universals that have been assigned.

In general, let the set of clauses not satisfied by an existential be C_u. As long as all the clauses in C_u are satisfied by some universal assignment, the remaining clauses can be satisfied by the existential assignment E_1. The only cases to be considered are those in which one or more of the clauses in C_u are not satisfied by any universal assignment. The new contribution of complete solution learning methods is to learn that the formula is satisfied so long as every clause in C_u is satisfied by a universal variable. We now describe how we can achieve this in one combined learning operation which does not increase the formula size exponentially.

Let U_i be the ith clause with the existentials removed. Let $C_u = \{c_p, c_q, \ldots\}$. We are to learn that so long as $L_n = U_p \wedge U_q \wedge \ldots$ is true, the original formula is true. L_n is not in DNF, making it hard to use standard learning methods. Note that in QBF learning, the original formula is true iff the DNF formula is true. Therefore, the QBF solver tries to make the DNF false to ensure that only new parts of the tree are explored. To put L_n in DNF, we replace each U_i by an 'indicator' variable I_i, using a simple encoding trick. We encode that a variable I_i is assigned true if and only if one of the universal variables in clause i satisfies the clause. This is encoded as several DNF cubes which can be added to the original formula before search begins. If $U_i = (u_1 \vee u_2 \vee \ldots \vee u_n)$, the cubes added will be $(\overline{u}_1 \wedge \overline{u}_2 \wedge \ldots \wedge \overline{u}_n \wedge I_i) \vee (u_1 \wedge \overline{I}_i) \vee (u_2 \wedge \overline{I}_n) \vee \ldots \vee (u_n \wedge \overline{I}_i)$. This formula shall be known as the indicator formula. So $L_n = I_p \wedge I_q \wedge \ldots$ can be added as a DNF cube when learning is performed. DNF unit propagation attempts to make the DNF formula false, and so the correct assignments are made. The indicator variables must be quantified after the variables in U_i, so that it is not possible to assign them before the variables which they indicate. The indicator variables must be universally quantified, as otherwise the cubes in the indicator formula would all be single-universal cubes. This would mean that the universals in the cubes would be assigned by propagation. The assignments of the correct value to the indicator variables is done by unit propagation in the DNF formula [10].

In solution learning, it is guaranteed that the addition of the DNF cubes does not affect the satisfiability of the formula i.e. $C \iff C \vee D$ where C is the original CNF formula and D is the additional DNF formula. If all the universal literals in U_i are assigned false, this removes all the binary cubes in the indicator formula and forces the indicator to be false by DNF unit propagation. If one of the universal literals in U_i is true, the indicator is assigned true by DNF unit propagation. This removes the remaining cubes. Thus the indicator formula can never make the formula true by itself. When a learned cube has been added, the formula can be made true by satisfying the learned cube. This will only occur if the universal variables are assigned so as to force the indicators to the values required to satisfy the learned cube. This will mean that the assignments to the universals are such that the current state has been visited previously and so no further search is required.

Solution backjumping must be slightly modified in order to support a solution set built from indicator variables. In building a solution set, where a clause is not satisfied by an existential, the indicator is added to the solution set instead of choosing one of the universals. Thus the final solution set will represent all the equivalent solution sets for the current assignment of variables. The resulting technique for backjumping is very similar to that of conflict-directed backjumping, and is shown in the context of our local complete learning algorithm in figure 4.

Unlike standard solution learning, complete solution learning does not guarantee to use the learned cube to invert the assignment to the universal. When the universal is assigned it may cause more than one indicator to be assigned through DNF unit propagation. This means that there is a possibility that more than one unassigned indicators occur in the learned cube when it is added. So the learned cube will not necessarily be a unit or single universal cube when it is added. Therefore, the algorithm must not rely on DNF unit propagation to reassign the universal after backtracking.

Take the example $\forall u_1 \forall u_2 \ldots \forall u_n \exists e_1 \ldots \exists e_m [(u_1 \vee u_2 \vee e_1) \wedge (u_3 \vee u_4 \vee e_2) \wedge (\overline{u}_4 \vee \overline{e}_2) \ldots]$. Before search begins, the formula is augmented with the indicator formula $(\overline{u}_1 \wedge \overline{u}_2 \wedge I_1) \vee (u_1 \wedge \overline{I}_1) \vee (u_2 \wedge \overline{I}_1) \vee (\overline{u}_3 \wedge \overline{u}_4 \wedge I_2) \vee (u_3 \wedge \overline{I}_2) \vee (u_4 \wedge \overline{I}_2) \vee (\overline{u}_4 \wedge \overline{I}_3) \vee (u_4 \wedge I_3)$. The indicators are quantified in a universal quantifier at the end of the formula. Now the assignments are made as follows: $u_1 = T, (I_1 = T), u_2 = T, u_3 = T, (I_2 = T), u_4 = F, (I_3 = F), E_1$, where E_1 is an assignment to the existentials such that $e_1 = F, e_2 = F \in E_1$. The assignments in brackets are those which are performed by unit propagation following the previous assignment. The solution set is built to contain $\{I_1, I_2\}$. Backjumping is performed to return to I_2. I_2 is removed and replaced with u_3. Backjumping then returns and unassigns u_3. The solution set now consists of $\{I_1\}$. Now the cube $(I_1 \wedge I_2)$ is learned, adding the indicators relevant to u_3 that were removed from the solution set during backtracking. As u_1 and u_2 are still assigned true, I_1 is true. I_2 must be assigned false by unit propagation. So u_3 and u_4 are assigned false by unit propagation. E_1 is no longer sufficient to satisfy the remaining formula so a new existential assignment must be found if the formula is to be proved true. So a new assignment E_2 is deduced such that $e_1 = F, e_2 = T \in E_2$. This results in the creation of the solution set $\{I_1, I_3\}$. Backjumping replaces I_3 with u_4 and returns to u_4. As both assignments have been made to u_4, the solution sets are combined to make $\{I_1\}$. Backjumping returns to the assignment of I_1 replacing it with u_1. The cube (I_1) is learned. This is a unit cube and thus I_1 is assigned false. This in turn assigns u_1 and u_2 false. This also removes the previously learned cube $(I_1 \wedge I_2)$. The algorithm is now free to choose values for u_3 and u_4.

General QBFs. The complete learning method described above only works when the formula prefix is of the form $\forall\exists$. If there is an existential quantified outside the universal, the learned cubes can become invalid. Our first solution to this is *local*: a learned cube can remain so long as no assignment to an existential quantified outside the universals indicated in the cube is changed. If such an existential assignment is changed, the learned cube is simply discarded.

If there is more than one level of universal quantification, the situation becomes further complicated. It becomes important to distinguish between the quantification levels of the universals. Take the quantifier sequence $\forall U_1 \exists E_2 \forall U_3 \exists E_4$ where U_1, E_2, U_3 and E_4 are sets of variables. The assignments to the variables in U_1 do not depend on any existential assignments. The assignments to the variables in U_3, however, depend on the assignments to the variables in E_2. Therefore, any locally learned cube that depends on a variable in U_3 must be deleted if a variable in E_2 is changed. A locally learned cube that only depends on variables from U_1 does not need to be deleted if E_2 is changed. Therefore we distinguish between the level of quantification of the information learned, by splitting the indicator variables for each clause. There is now one indicator variable for each level of universal quantification for each clause. The indicator variable $I_{i,j}$ is the indicator variable for the ith clause and the universals quantified in the jth quantifier. The rest of the procedure continues as before. Now, only universals that are quantified at the same level in a clause are considered to be equivalent. For example, take a QBF with a prefix $\forall u_1, u_2 \exists e_3 \forall u_4, u_5$ and a clause $(u_1 \vee u_2 \vee e_3 \vee u_4 \vee u_5)$. The indicator clauses added would be $(\overline{u}_1 \wedge \overline{u}_2 \wedge I_{1,1}) \vee (u_1 \wedge \overline{I}_{1,1}) \vee (u_2 \wedge \overline{I}_{1,1}) \vee (\overline{u}_4 \wedge \overline{u}_5 \wedge I_{1,2}) \vee (u_4 \wedge \overline{I}_{1,2}) \vee (u_5 \wedge \overline{I}_{1,2})$. So two indicators exist for the one clause.

When building a solution set, the algorithm may now be faced with two or more levels of universals to choose from. It makes sense to choose the outermost level of quantification when such a choice arises, since choosing the innermost level would mean that the variables in the outermost level would still satisfy the clause.

The main points of complete local solution learning are as follows:

1. For each clause, a disjunction of the universal literals with the same level of quantification are made equivalent to an indicator variable.
2. The indicator variables are all quantified in a universal quantifier at the end of the prefix.
3. All the indicator equivalences are encoded in a DNF formula and DNF unit propagation is used to enforce the equivalences.
4. When a solution is found, one indicator is put in the solution set for each clause not satisfied by an existential. This is the indicator for the outermost universal variable that satisfies the clause.
5. When backjumping is performed, and an indicator is unassigned, the indicator is replaced by one of the universals it indicates. This indicator is remembered.
6. When backjumping concludes with the reassignment of a universal, this universal is removed from the solution set and replaced by the indicators that are relevant to the universal and that were removed during backjumping. The solution set (consisting now only of indicators) is then added as a DNF cube.
7. When backjumping merges the solution sets for both assignments to a universal variable, all the indicators of the universal variable are removed from the resultant solution set.

Figures 1 - 4 show the important details of the algorithm. Figure 1 shows the top level of the algorithm. The main differences between this and the basic QBF algorithm are as follows. Firstly, the indicator formula must be generated. Secondly, as this is a local learning algorithm, any invalid cubes must be removed after backtracking. This

```
CompleteLocalSolutionLearn($PB$)
do
    result := propagate($P(B \vee D)$);
    if (result = UNDEF)
        (variable, value) := nextvariable($P(B \vee D)$);
    else if (result = TRUE)
        (variable, value) := CompleteLocalLearn($P(B \vee D)$);
    else if (result = FALSE)
        (variable, value) := falselearn($P(B \vee D)$);
        $S :=$ the set of learned cubes in D that contain indicators
            that indicate variables quantified inside variable;
        $D := D - S$;
    if (variable $\neq$ -1)
        $P(B \vee D)$ := $P(B \vee D)$[variable := value];
while (variable $\neq$ -1)
return result;
```

Note: the first line of the body is `$P(B \vee D)$ := GenerateIndicatorFormula($PB$);` and precedes `do`:

Fig. 1. The Algorithm for Complete Local Solution Learning. P is a prefix of quantifiers, B is a Boolean formula in CNF and D is a Boolean formula in DNF

```
CompleteLocalLearn($P(B \vee D)$)
$S$ := getIndicatorSet($P(B \vee D)$);
(variable, value, $R$) := clsbackjump($P(B \vee D)$, $S$);
$R' := \{r | r$ is an indicator of variable and $r \in R\}$;
$S := S \cup R$;
$L$ := set of learned cubes in $D$ that contain indicators
    that indicate variables quantified inside variable;
$D := D - L$;
$D := D + (\bigwedge_{s_i \in S} s_i)$;
return (variable, value);
```

Fig. 2. The algorithm for learning the solutions

```
getIndicatorSet($P(B \vee D)$)
if $D$ is true
    return the set of indicators in the cube that satisfies $D$;
else
    $S := \{\}$;
    for each clause $c$ in $B$
        if $c$ is not satisfied by an existential
            $v :=$ the outermost universal that satisfies the clause;
            $l :=$ the level of quantification of $v$;
            $I :=$ the indicator for clause $c$, level $l$;
            $S := S \cup \{I\}$;
    return $S$;
```

Fig. 3. The algorithm for calculating the indicator sets

is automatically handled by CompleteLocalLearn on a true backtrack but must be done explicitly after a false backtrack.

Figure 2 shows the details of the CompleteLocalLearn function. This calculates an indicator set, performs backjumping, then learns a new cube at the backtracking point. The learned cubes that are no longer valid after this backtrack are removed. Figure 3 shows the function for calculating the initial indicator set for backjumping. This is similar to calculating a solution set except that a universal variable chosen for the set is replaced by its indicator variable. Figure 4 shows how the indicator set is used by the solution directed backjumping procedure. For any variable assigned by unit propagation,

```
clsbackjump($P(B \vee D)$, $S$)
$R := \{\}$;
while there is backtracking to be done
   variable := the last assigned variable;
   $P(B \vee D) := P(B \vee D)$[variable unassigned];
   if variable $\in S$ and variable $\in \forall$
      if $l$ was assigned by single universal propagation
         $c :=$ the cube that caused the assignment of $l$;
         Add to every literal in $c$ to $S$;
         Remove variable from $S$;
         if variable is an indicator variable
            Add variable to $R$;
      else if both values of variable have been assigned
         $S' :=$ the indicator set from the last assignment of variable;
         $S := S \cup S'$;
      else
         value := value assigned to variable;
         return ($l$, value, $R$);
return (-1, false, {});
```

Fig. 4. Algorithm for backjumping using indicator sets. S is an indicator solution set

the variable is replaced by the variables in the cube that caused the variable to be assigned. Otherwise, if the variable has been backtracked to before, the indicator set obtained on the previous backtrack is joined with the new indicator set and the variable is removed. If the variable has not been backtracked on, it is now used as the backtrack point.

4 Complete Global Solution Learning

Local solution learning requires that learned information is thrown away when it is no longer valid. The use of only the universals in the solution sets means that the learned information can become invalid when some existentials are changed. The simplest way to overcome this problem is to add some existentials to the solution set. This is done in the same way as is done with solution learning. This way, the solutions can be kept beyond the reassignment of an existential. This means that the same solution will be avoided if it occurs later in the search. Take, for example, the formula $\exists e_1 \forall u_2, u_3 \exists e_4 \forall u_5, u_6 \exists e_7 [(e_4 \vee u_5 \vee u_6 \vee e_7) \wedge (\overline{e}_4 \vee \overline{u}_5) \wedge \ldots]$ and that there is an indicator $I_{1,4}$ for the first clause shown such that $I_{1,4} \iff u_5 \vee u_6$. Say that e_1, u_2, u_3, u_5 and u_6 are assigned true and that e_4 and e_7 are assigned false and that the remaining clauses are satisfied by the existentials. So now the solution set is built and $I_{1,4}$ is added to the set. Additionally, e_4 is added to the solution set as it satisfies the second clause. If e_1 does not satisfy any clause on its own then it does not need to be added to the solution set. So the cube learned is $(I_{1,4} \wedge \overline{e}_4)$. Now if later, e_4 is assigned true, this cube will be removed and so it cannot cause the DNF formula to become true. If later still, e_4 is unassigned, this cube will become active again. If e_4 is again assigned false, DNF unit propagation will assign $I_{1,4}$ false, resulting in u_5 and u_6 being assigned false. Hence, the same solution will not be explored again.

There is a second choice to make when considering the existentials that satisfy the clause. Only one existential is needed from each clause that is satisfied by an existential. In both normal solution learning and complete global solution learning, it makes sense to choose the innermost existential that satisfies a clause for the solution set when presented with a choice. Universals in a clause that are quantified inside all existentials in that clause can be removed [1]. Similarly, existentials in a cube that are quantified inside all universals in that cube can be removed [10]. By choosing the innermost existential that satisfies a clause at the same time as choosing the outermost universal where no existential satisfies a clause, there is more chance that the existential will be quantified inside the universals in the solution set. Therefore, the innermost existentials can be removed and so the added cube will be smaller and easier to manage.

There are reasons why local learning might be better than global learning. First, the learned information is dynamically deleted in local learning. Therefore, the size of the formula is less likely to grow exponentially in size. In global learning, learned information is kept for the duration of the search. However, two methods have been described [5] which dynamically delete the learned information. These are size bounded and relevance bounded learning. Local learning does not need these methods as it contains its own dynamic deletion system. If size and relevance bounding prove to be the best methods for controlling size growth of the formula then global learning will prove to be more useful. It may prove, however, that solutions do not reoccur outside the local branch being explored. Solutions depend on more information than conflicts in general; a conflict only requires one clause to become empty whereas a solution requires all clauses to be removed. Therefore, it is less likely that solutions will reoccur in different parts of the search tree, and local learning might prove to be the better learning method.

For space reasons we omit pseudocode for the global learning functions, as they are very similar to those for local learning. For global learning the algorithm is almost identical to that in figure 1. The main change is to call the global learning function CompleteGlobalLearn instead of CompleteLocalLearn on line 7. Also, the learned cubes are not discarded after an existential is reassigned, i.e. lines 10 & 11 are omitted. The global learning function CompleteGlobalLearn is also almost identical to CompleteLocalLearn shown in figure 2. As with the main function, the learned cubes are not removed: in this case we omit lines 5 & 6 of the previous function. Finally, the indicator set is built in the same way as for local learning, except now, if a clause is satisfied by an existential, the innermost existential that satisfies the clause is added to the solution set. That is, we omit lines 11, 12 & 13.

5 Experimental Evaluation

In the experiments presented here, the hypothesis will be clearly stated so that the experimental analysis is given more direction. On occasion, our original hypothesis will be wrong and corrected in discussion.

In all the implementations of algorithms presented, the same QBF solving library was used. The solving library performs all the basic tasks of the QBF solver. This includes reading the QBF instance from a file and initialising the data structures, assigning and unassigning variables and performing propagation steps. This avoids any differences between algorithms other than the intended differences. The library and all the solvers were written in C++ and compiled using GNU gcc 3.2. The compiler was passed the -O3 flag for optimisation of the code and the -static flag to statically link the libraries used. When a choice of variables is available the solvers all use the same heuristic, choosing the unassigned variable from the outermost quantifier which has the smallest variable index. The solvers were run on a cluster of Intel Pentium II 450 Mhz computers with 386 MB RAM running Redhat Linux 7.1. Each algorithm was given a timeout of 1200 seconds on each problem. The instances used were the benchmark instances available from QBFLib (www.qbflib.org) as of 15th March 2004.

Two types of graphs are presented, comparing the the number of backtracks of two algorithms, and comparing the run time. In a backtrack comparison, the number of backtracks performed by the base algorithm is plotted on the x-axis. The y-axis shows the improvement factor gained from using the second algorithm. This is calculated by dividing the number of backtracks performed by the algorithm without the new idea by the number performed by the algorithm with the new idea. A line is drawn at a value of 1 on the y-axis. This is called the equality line. Points which lie on this line are where both algorithms performed the same number of backtracks. A point which lies above the equality line is where the second algorithm performs less backtracks than the base algorithm on an instance. The distance of the point from the equality line indicates how much better or worse the algorithm with the new idea performed compared to the algorithm without the new idea. No points are plotted where either algorithm did not complete before the time-out. In a run time comparison, the graph is similar. This graph shows points where one algorithm timed out on an instance. Additionally, a diagonal line is drawn on the graph starting below the equality line and meeting the equality line at a value of 1200 seconds. This is known as the time-out line. A point which lies on this line is where the comparison algorithm timed-out but the base algorithm did not. Points which lie at 1200 seconds on the x-axis above the equality lines are ones where the base algorithm timed out but the comparison algorithm did not.

In order to evaluate the performance of the new solution learning techniques, each is compared to an algorithm that uses solution directed backjumping. Four QBF algorithms were implemented. These were CLearn, CSLearn, CLSLearn and CGSLearn. CLearn implements conflict learning and solution backjumping. CSLearn implements conflict and incomplete solution learning. CLSLearn implements conflict learning and complete local solution learning. CGSLearn implements conflict learning and complete global solution learning. We used CLearn and CSLearn as reference implementations, as they have all the same data structures, heuristics, etc, so run time and backtrack counts can be compared directly. In all comparisons, the number of solution backtracks is used. This is because the solution learning algorithms should directly affect the number of solution backtracks. The run time is also used in the comparisons. As new cubes are added to the QBF, more work must be done in assigning each variable. This work will directly affect the run time. It may not be worth doing the additional work in reducing the number of backtracks if more time is taken overall. In the complete learning techniques, some cubes are added during preprocessing. There may be a better way of setting up the indicators. An example of this is the use of an implicit assignment of the indicator variables. Therefore, the run time measure of these algorithms, whilst presented, does not necessarily reflect the best obtainable performance of the algorithm.

Our first experiments compared the existing and our new solution learning techniques with solution backjumping. All algorithms implement conflict learning.

Hypothesis

1. Solution learning performs the same or less solution backtracks than solution directed backjumping on a QBF benchmark instance.
2. Complete local solution learning performs the same or less solution backtracks than solution directed backjumping on a QBF benchmark instance.
3. Complete global solution learning performs the same or less solution backtracks than solution directed backjumping on a QBF benchmark instance.

Unfortunately, space precludes us from presenting our results in detail, but we summarise them as they inform the next set of experiments. Standard solution learning can perform less universal backtracks than solution backjumping. Unfortunately, there are very few points for which the difference between the algorithms is significant. As solution learning is just an overhead in most cases, using solution learning can result in the algorithm timing out where it would have completed if solution learning was not used. It is very rare when solution learning does complete faster than solution backjumping. Complete local solution learning performs better that solution backjumping on many problems. There are still many points for which complete local solution learning (CLSLearn) is an overhead overall. CLSLearn can sometimes solve problems that solution backjumping cannot within the timeout. Complete global solution learning performs better than solution backjumping on many problems. However, there are no points where complete global solution learning can solve problems that solution backjumping cannot but there are points that solution backjumping can solve that complete global solution learning cannot. It is unlikely that complete global solution learning is much better than incomplete solution learning when the run time is compared.

We are now more positive about complete local solution learning than in our previous report [4]. We ascribe this to more efficient implementation. This also suggests that performance of solution backjumping and learning techniques is critically dependent on small features of implementation, such as choice of solution sets in backjumping.

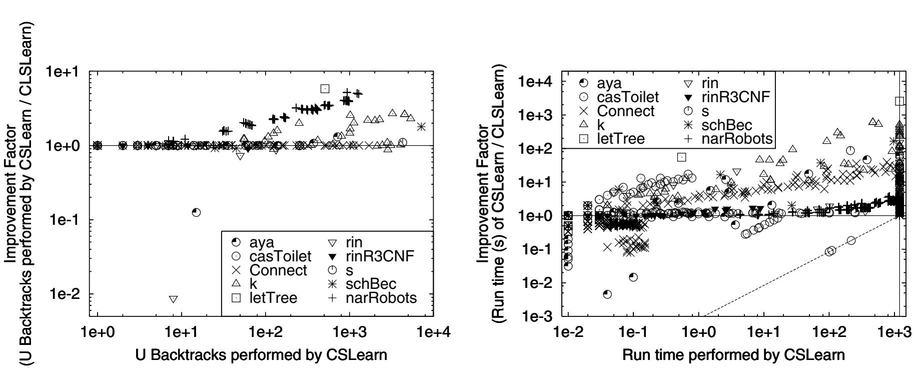


Fig. 5. A comparison of the number of universal backtracks (left) and run time (right) performed by solution learning and CLSLearn. For a description of the presentation of the graphs, see the main text

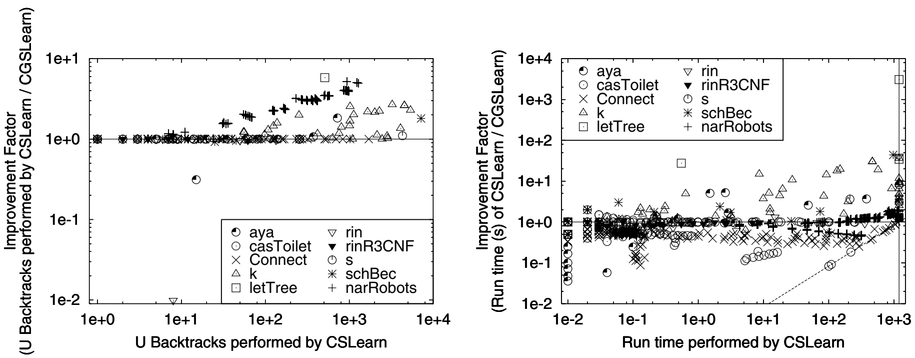


Fig. 6. A comparison of the number of universal backtracks (left) and run time (right) performed by solution learning and complete global solution learning

We next compare our new complete solution learning techniques, i.e. CLSLearn and CGSLearn, with the incomplete solution learning technique, i.e. CSLearn.

Hypothesis

1. Complete local solution learning performs the same or less universal backtracks than solution learning on a QBF benchmark instance.
2. Complete global solution learning performs the same or less universal backtracks than solution learning on a QBF benchmark instance.

Figure 5(left) shows that the improvement factor is often better than 1, though never reaching 10. Most points lie on or close to the equality line, with just 2 showing a significant deterioration. Figure 5(right) shows many points with an improvement factor better than one, and many that this enables CLSLearn to solve that CSLearn cannot. The best shows an improvement factor of over three orders of magnitude. There are three points that lie on the time-out line of CLSLearn.

Figure 6(left) is very similar to figure 5(left). Figure 6(right) shows a much smaller improvement in run time and many more cases where run time increases through the use of CGSLearn, compared to CLSLearn.

The results show that CLSLearn almost always performs the same or less universal backtracks than the incomplete solution learning technique. However, there are some points where this is not the case. This means that the first part of the hypothesis must be rejected. These points are likely to be caused by the CLSLearn technique missing out on finding a particularly good solution set and therefore performing more solution backtracks. As both the incomplete solution learning technique and CLSLearn both cause some overheads, the run time graph is much different to the comparison with solution backjumping. CLSLearn does cause some additional overheads compared to the incomplete solution learning technique. Therefore there are some instances that are not solved by CLSLearn but are solved by incomplete solution learning. There are many more points that can now be solved by CLSLearn that incomplete solution learning could not solve before the time-out. This shows the importance of learning more of the solutions in a solution learning technique.

The results for complete global solution learning (CGSLearn) are similar. Again, the second part of the hypothesis must be rejected. CGSLearn has even more overheads, as reflected in the run time comparison. There are more points where solution learning solves the instances but CGSLearn does not, and fewer points where CGSLearn solves the instance but incomplete solution learning does not. These points show that the complete learning technique is still worth doing. What is less obvious is whether it is worth performing global learning compared to performing local learning. This is explored in the next experiment.

Hypothesis

1. Complete global solution learning performs the same or less universal backtracks than complete local solution learning on a QBF benchmark instance.

Figure 7(left) shows some points with an improvement, but none with an improvement factor greater than 2.5. Figure 7(right) shows that the overheads cause most instances to deteriorate in run time, suggesting that CLSLearn is more effective than CGSLearn.

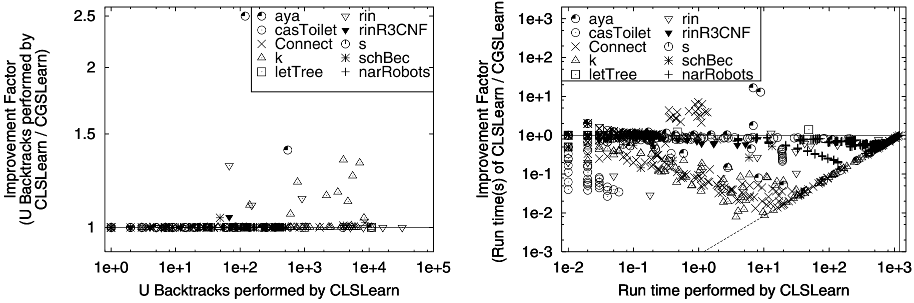


Fig. 7. A comparison of the number of universal backtracks (left) and run time (right) performed by CLSLearn and CGSLearn

The results show that the hypothesis is correct as CGSLearn never performs more universal backtracks than CLSLearn. However, only small improvements are obtained. As CGSLearn keeps the solutions for the remainder of the search, more overheads are caused than with CLSLearn. This is reflected in the run time where CGSLearn cannot solve many instances that CLSLearn can. There are no instances where CGSLearn solves the instance but CLSLearn does not. Therefore, it appears that it is not worth performing the global learning compared to local learning.

6 Conclusions

We presented two new techniques for solution learning. The first learns more information than would be learned by the incomplete solution learning technique. This is done without an exponential increase in formula size per solution despite the learning of an exponential number of solutions. This is achieved by using a simple but effective encoding trick. The second technique extends the first technique by allowing the learned solutions to be stored for the duration of the search.

Our experimental analysis shows that complete local solution learning performs better than the existing solution learning technique in most cases. Our second learning technique, complete global solution learning, does not perform as well, but still does better than the incomplete solution learning technique on many instances.

The implementations of the complete solution learning algorithms presented are not likely to be optimal. The encoding trick used in the solution learning adds to the original formula. This could be avoided by holding this information implicitly in the data structures of the QBF, or by only adding the indicator formula when it is needed. This may result in better operation of the complete solution learning techniques. It is worth first testing the hypothesis that the time taken to solve a QBF benchmark problem with indicator cubes is more than that taken to solve the QBF benchmark problem alone. This is likely to be the case since the indicator formula must be handled in addition to the original formula. This would also show how much more work must be done by the solver to deal with the indicator formula. It would also be interesting to try size and relevance bounding with the complete global solution learning technique.

Acknowledgements

The first author is supported by a Royal Society of Edinburgh SEELLD Support Fellowship and the second was supported by EPSRC grant GR/55382. We thank other members of the APES and CPPOD research groups, and Massimo Narizzano. We are very grateful to referees of SAT 05 for their helpful comments.

References

1. H. Kleine Buning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *information and Computation*, 117:12–18, 1995.
2. M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An algorithm to evaluate quantified Boolean formulae and its experimental evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.
3. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
4. I. Gent and A. Rowley. Solution learning and solution directed backjumping revisited. Technical Report APES-80-2004, APES Research Group, 2004. http://www.dcs.st-and.ac.uk/~apes/apesreports.html.
5. E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified Boolean logic satisfiability. In *Proc. AAAI 2002*, pages 649–654. AAAI Press, 2002.
6. E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for quantified Boolean logic satisfiability. *Artificial Intelligence*, 145(1–2):99–120, 2003.
7. R. Letz. Lemma and model caching in decision procedures for quantified Boolean formulas. In *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 160–175. Springer, 2002.

8. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.
9. L. Zhang and S. Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 442–449. ACM Press, 2002.
10. L. Zhang and S. Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In *Proc. CP 2002*, pages 200–215. Springer, 2002.

Equivalence Checking of Circuits with Parameterized Specifications

Eugene Goldberg

Cadence Berkeley Labs, USA
egold@cadence.com

Abstract. We consider the problem of equivalence checking of circuits N_1,N_2 with a common specification (CS). We show that circuits N_1 and N_2 have a CS iff they can be partitioned into toggle equivalent subcircuits that are connected "in the same way". Based on this result, we formulate a procedure for checking equivalence of circuits N_1 and N_2 with specifications S_1 and S_2. This procedure not only checks equivalence of N_1 and N_2 but also verifies that S_1 and S_2 are identical. The complexity of this procedure is linear in specification size and exponential in the value of a specification parameter. Previously we considered specifications parameterized by the size of the largest subcircuit (specification granularity). In this paper we give a more general parameterization based on specification "width".

1 Introduction

Studying equivalence checking (EC) of combinational circuits is of great practical and theoretical importance. A key problem of electronic chip design is to build a circuit that implements a Boolean function f and at the same time optimizes a cost function. In a typical design flow, this task is performed by a logic synthesis procedure. Such a procedure starts with some circuit N implementing f, and then gradually modifies N trying to improve the value of the cost function. Each optimization step requires EC to prove that the modified implementation of f is functionally equivalent to the previous one. Typically, due to high complexity of EC, a logic synthesis procedure uses only very simple transformations. So any discovery of a new powerful EC procedure will enable more powerful logic optimization steps thus drastically improving the quality of synthesized circuits.

Studying EC may be also very useful from a theoretical point of view for the following reasons. In terms of EC, the problem of checking if a circuit N is satisfiable (further referred to as Circuit-SAT) reduces to testing inequivalence of N and a trivial circuit $N(0)$ implementing constant 0. In [5] we showed that EC of circuits with a "similar" structure is easy. This result implies that it may be unreasonable to check for equivalence circuits N and $N(0)$. (Because N may have a very involved structure while $N(0)$ does not have any structure at all and so N and $N(0)$ are definitely not "similar".) Instead, one may try to build a sequence of circuits N_1,..., N_k where $N_1 = N$ and $N_k = N(0)$ such that EC of

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 107–121, 2005.

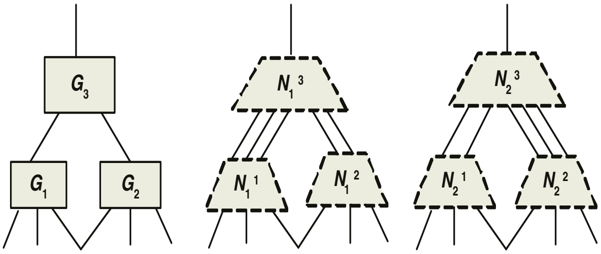


Fig. 1. Circuits N_1 and N_2 with a common specification of three mv-gates

a pair of consecutive circuits N_i,N_{i+1} is easy. Performing this kind of transitions from a "hard" circuit to an "easy" one requires a good understanding of EC complexity.

In [5] we studied EC of circuits with a common specification (CS). A CS S of N_1 and N_2 is just a circuit of multi-valued gates (mv-gates for short) such that N_1 and N_2 are different implementations of S. An example of a CS is given in Figure 1. Circuits N_1 and N_2 have a CS of 3 mv-gates shown on the left. Subcircuits N_1^i, N_2^i are different implementations of multi-valued mv-gate G_i of S. Circuit N_m^i (m=1,2) implements a multi-output Boolean function whose truth table is obtained from that of G_i by replacing values of multi-valued variables with their binary codes. So the difference between N_1^i and N_2^i is in the choice of binary encodings for the variables of S.

The main result of [5] is that, given circuits N_1 and N_2 and their CS represented by partitions N_1^1,.., N_1^k and N_2^1,.., N_2^k, there is a short "specification driven" resolution proof that N_1 and N_2 are equivalent. This proof is linear in the number of subcircuits k and exponential in size of the largest subcircuit N_i^j, i=1,2, j=1,..., k (granularity of specification). (The main point of [5] was to show that there is a very important class of practical formulas of very low "non-deterministic" complexity that are most likely hard for any deterministic resolution based SAT-solver.)

In this paper we develop further the theory of [5] and report the following new results.

- In [5] we gave an "explicit" definition of a CS S of circuits N_1 and N_2 where one needs to know not only the functionality of mv-gates of S but also the binary encodings of multi-valued variables. In this paper we show that N_1 and N_2 have a CS iff N_1 and N_2 can be partitioned into toggle equivalent subcircuits that are connected "in the same way". This result allows one to represent a CS of N_1 and N_2 "implicitly" as a partitioning of N_1 and N_2 into subcircuits.
- Based on the result above, we give a new procedure for EC of circuits with a CS.
- We give parameterization of our EC procedure in terms of specification width that generalizes the notion of specification granularity introduced in [5].

- We clarify the relation between fixed-parameter tractability and parameterized EC

The novelties of our EC procedure are as follows.

- In contrast to [5], when checking for equivalence of circuits N_1 and N_2 with specifications S_1 and S_2, our EC procedure does not assume that S_1 and S_2 are identical. Instead, it verifies the identity of S_1 and S_2 "on the fly".
- Our procedure works not only if circuits N_1 and N_2 are equivalent (as in [5]) but also when they are anti-equivalent (a special case of inequivalence).
- In contrast to [5], no restrictions are imposed on the length of binary encodings of multi-valued variables (in [5] we considered only minimal length encodings).

For lack of space, this paper does not contain any experimental results. A comparison of our EC procedure with a powerful industrial equivalence checker is given in [6]. The report [6] also describes application of our theory to logic synthesis. Namely, in [6] we formulate a powerful logic synthesis procedure that, given a circuit N_1 with a specification (represented as partition of N_1 into subcircuits), generates a circuit N_2 that implements the same specification as N_1. Today's state-of-the-art algorithms can neither generate such a circuit N_2 nor prove it to be equivalent to N_1.

This paper is structured as follows. In Section 2 the relation between fixed-parameter tractability and parameterized complexity of EC is discussed. In Section 3 we recall the notion of circuits with specifications. Section 4 introduces the notion of toggle equivalence of Boolean functions. In Section 5 we show that circuits N_1 and N_2 have a CS iff they can be partitioned into toggle equivalent subcircuits that are connected "in the same way" in N_1 and N_2. In Section 6 we give a procedure for EC of circuits N_1 and N_2 with predefined specifications. In Section 7 we discuss the complexity of our EC procedure and introduce width based parameterization of EC. In Section 8 we explain why it is hard to find a CS. Finally, we draw some conclusions in Section 9.

2 Fixed Parameter Tractability and Parameterized Complexity of Equivalence Checking

In this section, we discuss (very informally) the relation between fixed parameter tractability and parameterized complexity of equivalence checking. The notion of parameterized tractability [4] is an elegant way to form tractable subproblems of computationally hard problems. To parameterize a problem K means to build a function $\xi(K)$ that maps each instance I of K to an integer $p{=}\xi(I)$. A parameterized problem K is called fixed-parameter (FP) tractable if there is an algorithm that solves every instance I of K in time $\mathrm{O}(g(p) * L^m)$. Here $g(p)$ is an arbitrary function of p, L is the length of I, and m is a constant. It is not hard to see that any set of instances of the problem K that has the same value of p can be solved by this algorithm in time $\mathrm{O}(L^m)$. (The same applies to any set of instances of K that may have different values of p but the number of these values is finite.)

The EC procedure we give in this paper implies that EC has features of an FP tractable problem. Let K be the EC problem and I be an instance of K that is to check Boolean circuits N_1 and N_2 for equivalence. The function $\xi(K)$ is defined as follows. If N_1 and N_2 are functionally equivalent or anti-equivalent, then $p = \xi(I)$ is the width of a CS of N_1 and N_2 (see Section 7 for the definition). (Anti-equivalence means that Boolean functions implemented by N_1 and N_2 are complements of each other). As we will see if N_1 and N_2 are inequivalent but not anti-equivalent, then they do not have a CS. In that case, the value of $\xi(I)$ is arbitrary. Our EC procedure tests equivalence or anti-equivalence of N_1, N_2 or rejects the instance I (in case specifications of N_1 and N_2 are not identical) in time $\mathrm{O}(g(p) * L)$ where L is the instance length. A major difference between parameterized EC and an FP tractable problem is that our EC procedure needs an "advice". This advice is given in the form of specifications $S_1 = \{N_1^1, .., N_1^k\}$ and $S_2 = \{N_2^1, .., N_2^k\}$ represented as partitions of N_1 and N_2 into subcircuits.

Let us explain the relation between parameterized EC and FP tractability by the example of Circuit-SAT. Circuit-SAT is an NP-complete problem which is to test if a circuit N is satisfiable. Circuit-SAT is known to be an FP tractable problem (see for example [3, 8]. The satisfiability of N can be solved in time $\mathrm{O}(exp(c * w) * |N|)$ where w is the circuit width, c is a constant and $|N|$ is the circuit size. (We do not clarify which definition of circuit width we use because the discussion below applies to any of them.) The quick growth of $exp(c * w)$ makes it possible to efficiently solve Circuit-SAT only for very "narrow" circuits. On the other hand, our EC procedure can be efficiently applied to circuits N_1 and N_2 of arbitrary width. The limiting factor is the width of each pair N_1^i, N_2^i, $i = 1, .., k$ of corresponding subcircuits of S_1 and S_2. In other words, in the case of Circuit-SAT, the parameter describes the **"absolute"** circuit width. In the case of EC the parameter describes the **"differential"** width of circuits N_1 and N_2 that indicates how different N_1 and N_2 are.

One can view width-based tractability of Circuit-SAT as a "special case" of width-based tractability of EC. Indeed, on the one hand, testing the satisfiability of a circuit reduces to testing inequivalence of this circuit to constant 0. On the other hand, if N_2 is a trivial implementation of constant 0, then the relative width of N_1 and N_2 is actually the absolute width of N_1. While FP tractability looks for circuits that are "immediately" simple, parameterized EC implies the possibility of making circuits simpler "gradually" by performing "narrow" (and so easily verifiable) changes. For example, if the absolute width of N_2 is smaller than that of N_1 and the width of their CS is small, then one can replace $Circuit\_SAT(N_1)$ with $Circuit\_SAT(N_2)$ easily proving that this replacement is correct.

3 Boolean Circuits with Specifications

In this section, we recall the “old” definition of a CS of two circuits N_1 and N_2 given in [5]. In Section 5 we will give a new definition allowing to represent CS “implicitly” as partitioning of N_1 and N_2 into toggle equivalent subcircuits. We will also show that these two definitions are equivalent.

An mv-circuit (here ***mv*** stands for ***multi-valued***) is a network of ***mv-gates***, exactly as a Boolean circuit is a network of Boolean gates. An mv-gate of n inputs is a device that implements an mv-function of n mv-variables. (Similarly, a Boolean gate of n inputs is a device implementing a Boolean function of n Boolean variables.)

We will call a Boolean function $\boldsymbol{f}$ (or a Boolean circuit implementing $\boldsymbol{f}$) specifying a mapping $\{0,1\}^m \rightarrow \{0,1\}^p$ a ***multi-output function*** (a multi-output circuit respectively) if $p \geq 1$ and a ***single-output function*** (a single output circuit respectively) if $p = 1$. Let X be a multi-valued variable. Let $\boldsymbol{q}(X)$ be a Boolean function mapping each value X' of X to a point of $\{0,1\}^n$. The function $\boldsymbol{q}(X)$ is called an (***n-bit***) ***encoding*** of X if different values X' and X'' of X are mapped to different points i.e. $\boldsymbol{q}(X') \neq \boldsymbol{q}(X'')$. The value of $\boldsymbol{q}$ at X' is called the ***code*** of X'.

Let $G(X_1,\ldots,X_n)$ be the mv-function implemented by an mv-gate G and Y be the mv-variable describing the output of G. A Boolean function $\boldsymbol{f}$ ***implements*** G if there are n+1 Boolean encodings $\boldsymbol{q}_1(X_1)$,$\boldsymbol{q}_2(X_2)$,...,$\boldsymbol{q}_n(X_n)$,$\boldsymbol{q}(Y)$ such that a) $Y = G(X_1,.., X_n)$ implies $\boldsymbol{q}(Y)$=$\boldsymbol{f}(\boldsymbol{q}_1(X_1),..,\boldsymbol{q}_n(X_n))$ and b) each combination of output values produced by $\boldsymbol{f}$ is a code of a value of Y. If a Boolean circuit N implements a Boolean function $\boldsymbol{f}$ implementing an mv-gate G, we will say that N implements G.

A multi-valued circuit S of k gates G_1,..,G_k is called a ***specification*** of a Boolean circuit N if N is a composition of k subcircuits N_1,..,N_k such that a) subcircuit N_i is an implementation of mv-gate gate G_i; b) subcircuits of N are connected as corresponding gates of S (for example, the circuit N_1 shown in the center of Figure 1 consists of three subcircuits N_1^1, N_1^2, N_1^3 connected as the corresponding mv-gates G_1,G_2,G_3 of the specification shown on the left.)

Definition 1. *An mv-circuit S is called a* ***common specification (CS)*** *of Boolean circuits N_1 and N_2 if S is a specification of both N_1 and N_2.*

In this paper we make the following ***assumptions***.

1. Each gate of a Boolean circuit has only two inputs (no assumptions are made about the number of inputs of an mv-gate).
2. Circuits N_1 and N_2 to be proven equivalent have only one output.
3. Every specification has only one output and this output takes only Boolean values. Every specification has only Boolean inputs. (This means, for example, that the inputs of gates G_1 and G_2 and the output of gate G_3 in Figure 1 take only Boolean values.) In other words, only “internal” variables of a specification can be multi-valued. All the “external” variables are Boolean.

4 Toggle Equivalence of Boolean Functions

According to Definition 1, to show that circuits N_1, N_2 have a CS one has to present an mv-circuit S and binary encodings of the mv-gates of S that prove N_1 and N_2 to be implementations of S. In Sections 4 and 5 we show that one can represent a CS of N_1 and N_2 "implicitly" as a partitioning of N_1 and N_2 into toggle equivalent subcircuits. In this case, no explicit knowledge of encodings or the functionality of mv-gates is necessary.

In this section, we introduce the notion of toggle equivalence. We show that toggle equivalent Boolean functions can be considered as different implementations of the same multi-valued function.

4.1 Toggle Equivalence of Functions with Identical Sets of Variables

Definition 2. *Let $\boldsymbol{f}_1$:$\{0,1\}^n \rightarrow \{0,1\}^m$ and $\boldsymbol{f}_2$:$\{0,1\}^n \rightarrow \{0,1\}^k$ be m-output and k-output Boolean functions of the same set of variables. Functions $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are called* ***toggle equivalent*** *if $\boldsymbol{f}_1(\boldsymbol{x}) \neq \boldsymbol{f}_1(\boldsymbol{x'}) \Leftrightarrow \boldsymbol{f}_2(\boldsymbol{x}) \neq \boldsymbol{f}_2(\boldsymbol{x'})$. Circuits N_1 and N_2 implementing toggle equivalent functions $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are called* ***toggle equivalent circuits****.*

Informally, toggle equivalence means that for any pair of input vectors $\boldsymbol{x}$,$\boldsymbol{x'}$ for which at least one output of N_1 "toggles", the same is true for N_2 and vice versa.

Definition 3. *Let $\boldsymbol{f}$ be a multi-output Boolean function of n variables. Denote by Part($\boldsymbol{f}$) the partition of the set $\{0,1\}^n$ into disjoint subsets $B_1,\dots,B_k$ such that $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x'})$ iff $\boldsymbol{x}$, $\boldsymbol{x'}$ are in the same subset B_i.*

Proposition 1. *Two Boolean functions $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are toggle equivalent iff Part($\boldsymbol{f}_1$)=Part($\boldsymbol{f}_2$) i.e. iff for each element B_i of the partition Part($\boldsymbol{f}_1$) there is an element B'_j of the partition Part($\boldsymbol{f}_2$) such that B_i=B'_j and vice versa.*

Proof. If $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are toggle equivalent, there cannot exist a pair of vectors $\boldsymbol{x}$,$\boldsymbol{x'}$ such that $\boldsymbol{x}$,$\boldsymbol{x'}$ are in the same subset of one partition and in different subsets of the other partition. (Because that would mean that one function produces two identical output assignments while the other function toggles.)

Proposition 2. *Let $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ be toggle equivalent single output Boolean functions. Then $\boldsymbol{f}_1$=$\boldsymbol{f}_2$ or $\boldsymbol{f}_1$=$\overline{\boldsymbol{f}}_2$ where $\overline{\boldsymbol{f}}_2$ is the negation of $\boldsymbol{f}_2$.*

Proof. From Proposition 1 it follows that $Part(\boldsymbol{f}_1)$=$Part(\boldsymbol{f}_2)$. Since $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are single output Boolean functions, $Part(\boldsymbol{f}_1)$ and $Part(\boldsymbol{f}_2)$ consist of two elements each. So $\boldsymbol{f}_1$=$\boldsymbol{f}_2$ or $\boldsymbol{f}_1$=$\overline{\boldsymbol{f}}_2$.

Proposition 3. *Let $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ be toggle equivalent. Then $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are two different implementations of the same multi-valued function of Boolean variables.*

Proof. According to Proposition 1, $Part(\boldsymbol{f}_1)$=$Part(\boldsymbol{f}_2)$. Let $Part(\boldsymbol{f}_1)$, $Part(\boldsymbol{f}_2)$ consist of k elements each. Then $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are implementations of the function F: $\{0,1\}^n \rightarrow \{1,..,k\}$ where $F(\boldsymbol{x})$=m, iff $\boldsymbol{x} \in B_m$, i.e. iff $\boldsymbol{x}$ is in the m-th element of $Part(\boldsymbol{f}_1)$.

4.2 Toggle Equivalence of Functions with Different Sets of Variables

In this subsection, the notion of toggle equivalence is extended to the case of Boolean functions with different sets of variables that are related by constraint functions.

Definition 4. *Let X and Y be two disjoint sets of Boolean variables (the number of variables in X and Y may be different). A function $Cf(X,Y)$ is called a* ***correlation function*** *if there are subsets $A^X \subseteq \{0,1\}^{|X|}$ and $A^Y \subseteq \{0,1\}^{|Y|}$ such that*

1. *$|A^X| = |A^Y|$ and*
2. *$Cf(X,Y)$ specifies a bijective mapping M: $A^X \rightarrow A^Y$. Namely $Cf(\boldsymbol{x}, \boldsymbol{y})=1$ iff $\boldsymbol{x} \in A^X$ and $\boldsymbol{y} \in A^Y$ and $\boldsymbol{y} = M(\boldsymbol{x})$.*

Remark 1. Informally, $Cf(X,Y)$ is a correlation function if it specifies a bijective mapping between a subset A^X of $\{0,1\}^{|X|}$ and a subset A^Y of $\{0,1\}^{|Y|}$. So one can view $Cf(X,Y)$ as relating two different encodings of an $|A^X|$-valued variable.

As Proposition 4 below shows, one can check if a Boolean function $H(X,Y)$ is a correlation one without explicitly finding subsets A^X and A^Y.

Proposition 4. *Let X and Y be two disjoint sets of Boolean variables. A Boolean function $H(X,Y)$ is a correlation one iff the following two conditions hold.*

1. *There do not exist three vectors $\boldsymbol{x}$, $\boldsymbol{x'}$,$\boldsymbol{y}$ (where $\boldsymbol{x}$, $\boldsymbol{x'}$ are assignments to variables of X and $\boldsymbol{y}$ is an assignment to variables of Y) such that $\boldsymbol{x} \neq \boldsymbol{x'}$ and $H(\boldsymbol{x},\boldsymbol{y})$=$H(\boldsymbol{x'},\boldsymbol{y})$=1.*
2. *There do not exist three vectors $\boldsymbol{x}$,$\boldsymbol{y}$,$\boldsymbol{y'}$ such that $\boldsymbol{y} \neq \boldsymbol{y'}$ and $H(\boldsymbol{x},\boldsymbol{y})$= $H(\boldsymbol{x},\boldsymbol{y'})$=1.*

Proof. Only if part. If $H(X,Y)$ is a correlation function, the fact that conditions 1) and 2) hold, follows from Definition 4.

If part. If conditions 1) and 2) hold then, $H(X,Y)$ specifies a bijective mapping between subsets A^X and A^Y defined in the following way. Subset A^X consists of all the assignments $\boldsymbol{x} \in \{0,1\}^{|X|}$ such that $H(\boldsymbol{x}, \boldsymbol{y})$=1 for some $\boldsymbol{y} \in \{0,1\}^{|Y|}$. Subset A^Y consists of all the assignments $\boldsymbol{y} \in \{0,1\}^{|Y|}$ such that $H(\boldsymbol{x}, \boldsymbol{y})$=1 for some $\boldsymbol{x} \in \{0,1\}^{|X|}$.

Remark 2. Checking if $H(X,Y)$ is a correlation function reduces to solving two instances of the satisfiability problem (SAT). Checking condition 1) of Proposition 4 reduces to testing the satisfiability of the expression $H(X,Y) \wedge H(X',Y')$ $\wedge$ $Neq(X,X') \wedge Eq(Y,Y')$. Here $H(X',Y')$ is a "copy" of $H(X,Y)$ where variables of X',Y' are independent of those of X,Y. $Neq(\boldsymbol{x},\boldsymbol{x'})$ is equal to 1 iff $\boldsymbol{x} \neq \boldsymbol{x'}$. Function $Eq(Y,Y')$ is the negation of $Neq(Y,Y')$. Checking condition 2) of Proposition 4 reduces to testing the satisfiability of $H(X,Y) \wedge H(X',Y')$ $\wedge$ $Eq(X,X') \wedge Neq(Y,Y')$. If either expression is constant 0 (i.e. unsatisfiable), then H is a correlation function.

Definition 5. *Let* $\boldsymbol{f}_1$*:*$\{0,1\}^n \rightarrow \{0,1\}^m$ *and* $\boldsymbol{f}_2$*:*$\{0,1\}^p \rightarrow \{0,1\}^k$ *be m-output and k-output Boolean functions and X and Y specify their sets of Boolean variables where* $|X| = n$ *and* $|Y| = p$*. Let* $D_{inp}(X,Y)$ *be a Boolean function. Functions* $\boldsymbol{f}_1$ *and* $\boldsymbol{f}_2$ *are called* ***toggle equivalent under constraint function*** $D_{inp}(X,Y)$ *if* $(\boldsymbol{f}_1(\boldsymbol{x}) \neq \boldsymbol{f}_1(\boldsymbol{x'}) \wedge (D_{inp}(\boldsymbol{x},\ \boldsymbol{y}) = D_{inp}(\boldsymbol{x'},\boldsymbol{y'}) = 1)) \Rightarrow (\boldsymbol{f}_2(\boldsymbol{y}) \neq \boldsymbol{f}_2(\boldsymbol{y'})$ *and vice versa* $(\boldsymbol{f}_2(\boldsymbol{y}) \neq \boldsymbol{f}_2(\boldsymbol{y'}) \wedge (D_{inp}(\boldsymbol{x},\boldsymbol{y}) = D_{inp}(\boldsymbol{x'},\boldsymbol{y'}) = 1)) \Rightarrow \boldsymbol{f}_1(\boldsymbol{x}) \neq \boldsymbol{f}_1(\boldsymbol{x'})$*.*

Proposition 5. *Let X,Y be sets of Boolean variables and* $\{X_1,\ldots,X_s\}$ *and* $\{Y_1,\ldots,Y_s\}$ *be partitions of X and Y respectively. Let* $Cf(X_1,Y_1),..,Cf(X_s,Y_s)$ *be correlation functions. Let* $\boldsymbol{f}_1(X)$ *and* $\boldsymbol{f}_2(Y)$ *be toggle equivalent under the constraint function* $D_{inp}(X,Y) = Cf(X_1,Y_1) \wedge \ldots \wedge Cf(X_s,Y_s)$*. Then* $\boldsymbol{f}_1$ *and* $\boldsymbol{f}_2$ *are implementations of the same multi-valued function of s multi-valued variables.*

Proof follows from Proposition 3 and Remark 1.

4.3 Testing Toggle Equivalence

In this subsection, we show how to test toggle equivalence of multi-output Boolean circuits N_1 and N_2. Namely we show that testing the toggle equivalence of N_1 and N_2 reduces to checking if function $D_{out}(N_1,\ N_2)$ specified by Definition 8 (see below) is a correlation one. This test can be performed by two satisfiability checks as described in Remark 2.

Definition 6. *Let N be a Boolean circuit. Denote by* $v(N)$ *the set of Boolean variables associated with the output or an input of a gate of N. Denote by* ***Sat(v(N))*** *the Boolean function such that Sat(z)=1 iff the assignment z to variables* $v(N)$ *is "possible" i.e consistent. For example, if circuit N consists of just one AND gate* $y = x_1 \wedge x_2$*, then* $v(N) = \{y,\ x_1,\ x_2\}$ *and* $Sat(v(N)) = (\overline{x}_1 \vee \overline{x}_2 \vee y) \wedge (x_1 \vee \overline{y}) \wedge (x_2 \vee \overline{y})$*.*

Definition 7. *Let f be a Boolean function. We will say that function* f^* *is obtained from f by* ***existentially quantifying away variable*** *x if* $f^* = f(\ldots,\ x{=}0,\ldots) \vee f(\ldots,\ x{=}1,\ldots.)$*.*

If f is represented as a CNF formula F, then to existentially quantify away a variable x one just needs to add to F all the clauses produced by resolving clauses of F in x and to remove from F all the clauses having a literal of x.

Definition 8. *Let* N_1 *and* N_2 *be Boolean circuits whose inputs are specified by sets of variables X and Y respectively. Let* $D_{inp}(X,Y)$ *be a Boolean function. Denote by* $\boldsymbol{D_{out}(N_1,\ N_2)}$ *the Boolean function obtained from the Boolean function H, where* $H = Sat(v(N_1)) \wedge Sat(v(N_2)) \wedge D_{inp}(X,Y)$*, by existentially quantifying away all the variables of H but the output variables of* N_1 *and* N_2 *(i.e. the variables associated with outputs of* N_1 *and* N_2*).*

Proposition 6. *Let* N_1 *and* N_2 *be Boolean circuits with input variables specified by sets X, Y respectively. Let* $D_{inp}(X,Y)$ *be a Boolean function relating X and Y. Let* $D_{inp}(X,Y)$ *be a correlation function. Then* N_1 *and* N_2 *are toggle equivalent under constraint function* $D_{inp}(X,Y)$ *iff the function* $D_{out}(N_1,N_2)$ *specified in Definition 8 is also a correlation function.*

Proof. Only If part. Let N_1 and N_2 be toggle equivalent under constraint function $D_{inp}(X,Y)$. Then $D_{out}(N_1,N_2)$ satisfies either condition of Proposition 4 and hence it is a correlation function. For example, there cannot exist Boolean vectors $\boldsymbol{z}$, $\boldsymbol{z'}$ and $\boldsymbol{h}$ (where $\boldsymbol{z} \neq \boldsymbol{z'}$ and $\boldsymbol{z}$, $\boldsymbol{z'}$ are output assignments of N_1 and $\boldsymbol{h}$ is an output assignment of N_2) such that $D_{out}(\boldsymbol{z},\boldsymbol{h})$=$D_{out}(\boldsymbol{z'},\boldsymbol{h})$=1. Indeed, it would mean that there exist pairs of vectors $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{x'}$, $\boldsymbol{y'}$ such that a) $\boldsymbol{z}$=$N_1(\boldsymbol{x})$, $\boldsymbol{z'} = N_2(\boldsymbol{x'})$ and $\boldsymbol{h} = N_2(\boldsymbol{y})$=$N_2(\boldsymbol{y'})$;b) $D_{inp}(\boldsymbol{x}, \boldsymbol{y})$=1 and $D_{inp}(\boldsymbol{x'}, \boldsymbol{y'})$=1; c) $\boldsymbol{x} \neq \boldsymbol{x'}$ and $\boldsymbol{y} \neq \boldsymbol{y'}$; d) $N_1(\boldsymbol{x}) \neq N_1(\boldsymbol{x'})$ while $N_2(\boldsymbol{y}) = N_2(\boldsymbol{y'})$. But this is impossible because N_1 and N_2 are toggle equivalent.

If part can be proven in a similar manner.

5 Common Specification and Toggle Equivalence

In this section, we show that the existence of a CS of single output combinational circuits N_1 and N_2 means that N_1, N_2 can be partitioned into toggle equivalent subcircuits that are connected in N_1 and N_2 "in the same way". This result is formulated in Proposition 7.

Definition 9. *Let N= (V,E) be a directed acyclic graph (**DAG**) representing a Boolean circuit. (Here V,E are sets of nodes and edges of N respectively.) A subgraph N^*=(V^*, E^*) of N is called a **subcircuit** if the following two conditions hold:*

1. *if g_1, g_2 are in V^* and there is a path from g_1 to g_2 in N, then all the nodes of N on that path are in V^* ;*
2. *if g_1, g_2 of V^* are connected by an edge in N, then they are also connected by an edge in N^*.*

Definition 10. *Let N^* be a subcircuit of N. An input of a gate g of N^* is called **an input** of N^* if it is not connected to the output of some other gate of N^*. The output of a gate g is called **an output** of subcircuit N^* if a) it is the output of N or b) it is connected to an input of a gate of N that is not in N^*.*

Definition 11. *Let a Boolean circuit N be partitioned into k subcircuits N^1, ..., N^k . Let T be a directed graph of k nodes such that nodes G_i and G_j of T are connected by a directed edge (from G_i to Gj) iff an output of N^i is connected to an input of N^k in N. T is called **the communication specification** corresponding to the partition N^1, ..., N^k. The partition N^1, ..., N^k is called **topological** if T is a DAG (i.e. if T does not contain cycles).*

Definition 12. *Let T be the communication specification of circuit N with respect to a topological partition N^1, ..., N^k. Let G_i be the node of T corresponding to subcircuit N^i. The length of the longest path from an input of T to G_i is called the **level** of G_i and N^i (denoted by level(G_i) and level(N^i) respectively).*

Definition 13. *Let N_1^1, ...,N_1^k and N_2^1, ..., N_2^k be topological partitions of single output Boolean circuits N_1,N_2. Let communication specifications of N_1 and*

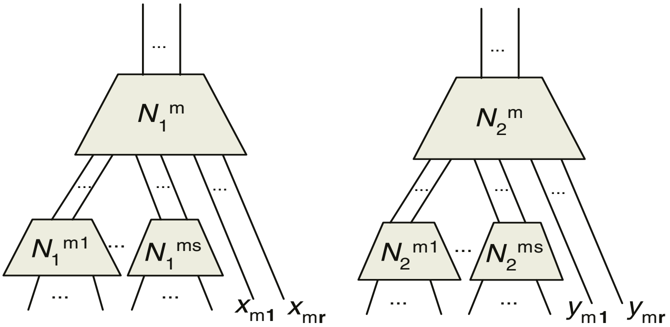


Fig. 2. Illustration to Definition 13

*N_2 with respect to partitions N_1^1, ..., N_1^k and N_2^1, ..., N_2^k be identical. Denote by **$D_{out}(N_1^m,N_2^m)$, m=1,. . . ,k** functions computed by induction in topological levels. Namely, we first compute functions D_{out} for the subcircuits of level 1, then for those of level 2 and so on. Function $D_{out}(N_1^m,N_2^m)$ is obtained from function H_m=Sat($v(N_1^m)$) $\wedge$ Sat($v(N_2^m)$) $\wedge D_{inp}(N_1^m,N_2^m)$ by existentially quantifying away all the variables except the output variables of N_1^m, N_2^m. The function $D_{inp}(N_1^m,N_2^m)$ is equal to $D_{out}(N_1^{m1},N_2^{m1})\wedge$ .... $\wedge D_{out}(N_1^{ms},N_2^{ms})\wedge$ Eq(x_{m1},y_{m1})$\wedge$... $\wedge$ Eq(x_{mr},y_{mr}). Here N_1^{m1},..,N_1^{ms},N_2^{m1},..,N_2^{ms} are the s subcircuits (if any) whose outputs are connected to inputs of N_1^m,N_2^m respectively. (See illustration in Figure 2.) Variables x_{m1},. . . ,x_{mr},y_{m1},.., y_{mr} are the r input variables of N_1 and N_2 (if any) that feed N_1^m and N_2^m respectively. Function Eq(x_{mt},y_{mt}), 1$\leq$t $\leq$ r is equal to 1 iff x_{mt} is equal to y_{mt}.*

Proposition 7. *Let N_1, N_2 be two single-output Boolean circuits and T be a DAG of k nodes. Circuits N_1 and N_2 are implementations of a specification S whose topology is given by T iff there is a partition Spec(N_1) = $\{N_1^1$, ..., $N_1^k\}$ of N_1 and a partition Spec(N_2) = $\{N_2^1$, ..., $N_2^k\}$ of N_2 into k subcircuits such that*

- *Communication specifications T_1,T_2 of N_1 and N_2 with respect to partitions Spec(N_1), Spec(N_2) are equal to T;*
- *Each pair of circuits N_1^m, N_2^m is toggle equivalent under constraint function $D_{inp}(N_1^m, N_2^m)$ specified by Definition 13.*

Sketch of the proof. If part. It is proven by induction (in levels) using Proposition 5 and Proposition 6. ***Only if part*** is proven by induction using the fact that two Boolean functions implementing the same multi-valued function are toggle equivalent.

Proposition 7 allows one to specify a CS S of circuits N_1 and N_2 “implicitly” by giving partitions $\{N_1^1$,.., $N_1^k\}$ and $\{N_2^1,\ldots, N_2^k\}$. Of course, knowing these partitions one can always obtain the functionality of all mv-gates of S and find the binary encodings using which circuits N_1 and N_2 can be produced from S.

From Proposition 7 it follows that if N_1 and N_2 have a CS, then N_1^k and N_2^k (i.e. the “last” subcircuits of N_1 and N_2 respectively) and hence N_1 and N_2 are toggle equivalent. From Proposition 2 it follows then that N_1 and N_2 are either equivalent or anti-equivalent. In other words, if N_1 and N_2 are neither equivalent nor anti-equivalent, they have no CS.

6 A Procedure for Equivalence Checking of Circuits with a Common Specification

In this section, we describe a procedure for EC of circuits N_1, N_2 with predefined specifications S_1, S_2. We will refer to this procedure as ECCS (EC of circuits with a CS). The ECCS procedure essentially mimics a specification guided resolution proof introduced in [5]. However there are a few differences.

- The ECCS procedure does not assume that specifications S_1 and S_2 are identical. Instead, using Proposition 7, it checks if S_1 and S_2 are identical on the fly.
- The ECCS procedure not only proves that N_1 and N_2 are functionally equivalent but can also prove that N_1 and N_2 are anti-equivalent.
- The ECCS procedure is described in terms of functions and existential quantification independently of the way these functions are represented.

The pseudocode of the ECCS procedure is shown in Figure 3. The procedure *topol_partition* checks if *Spec*(N_1) and *Spec*(N_1) are topological partitions (see Definition 11). The procedure *equiv_commun_specs* checks if communication specifications T_1 of N_1 (with respect to *Spec*(N_1)) and T_2 of N_2 (with respect to *Spec*(N_2)) are identical.

In the main loop, functions $D_{out}(N_1^i, N_2^i)$ are computed in topological order as described in Definition 13. Before computing $D_{out}(N_1^i, N_2^i)$ the procedure *con-*

```
/* Spec(N_1)= {N_1^1,..,N_1^k},Spec(N_2)= {N_2^1,..,N_2^k} */
ECCS(N_1, N_2, Spec(N_1),Spec(N_2)) {
      if (topol_partition(N_1,N_2,Spec(N_1),Spec(N_2)) == 'no')
        return('reject');
      if (equiv_commun_specs( N_1,N_2,Spec(N_1),Spec(N_2)) == 'no')
        return('reject');
      for (i=1; i <= k ; i++) {
        D_inp = constr_func(N_1^i,N_2^i,N_1,N_2);
        D_out(N_1^i, N_2^i) = exist_quantify(N_1^i,N_2^i, D_inp);
        if (correlation_function(D_out) == 'no')
          return('reject');}

/* At this point we know that D_out(N_1^k, N_2^k) is a correlation function */
      if (D_out(N_1^k, N_2^k) implies equivalence of N_1 and N_2 )
        return('equivalent');
      if (D_out(N_1^k, N_2^k) implies anti-equivalence N_1 and N_2)
        return('anti-equivalent');}
```

Fig. 3. Pseudocode of the ECCS procedure

str_func forms the expression D_{inp} (see Definition 13). The *exist_quantify* procedure existentially quantifies away from the function $H_i{=}Sat(v(N_1^i)) \wedge Sat(v(N_2^i)) \wedge D_{inp}$ all the variables except the output variables of N_1^i and N_2^i. Then the *correlation_function* procedure checks if the result of quantification D_{out} is a correlation function. This check reduces to two SAT-checks as described in Remark 2.

Finally, the ECCS procedure checks if the correlation function of $D_{out}(N_1^k, N_2^k)$ relating outputs N_1^k and N_2^k (and so outputs of N_1 and N_2) implies that N_1 and N_2 are equivalent or anti-equivalent. If $D_{out}(N_1^k, N_2^k)$ is equal to $(\overline{z}_1 \vee z_2) \wedge (z_1 \vee \overline{z}_2)$ (here z_1 and z_2 are Boolean variables associated with the output of N_1^k and N_2^k respectively), then N_1 and N_2 are functionally equivalent. If $D_{out}(N_1^k, N_2^k)$ is equal to $z_1 \wedge z_2$ (respectively $\overline{z}_1 \wedge \overline{z}_2$) then N_1 and N_2 are functionally equivalent and implement constant 1 (respectivley 0). If $D_{out}(N_1^k, N_2^k)$ is equal to $(z_1 \vee z_2) \wedge (\overline{z}_1 \vee \overline{z}_2)$ or $z_1 \wedge \overline{z}_2$ or $\overline{z}_1 \wedge z_2$, then functions implemented by N_1 and N_2 are complements of each other. It is not hard to show that any Boolean function of z_1,z_2 different from the six functions above is not a correlation function.

ECCS procedure returns the '*reject*' answer if any of the checks performed by *topol_partition, equiv_commun_specs* and *correlation_function* fails. The rejection means that partitions *Spec*(N_1) and *Spec*(N_2) do not represent a CS of N_1 and N_2.

7 Parameterization of EC by Specification Width

In this section we discuss the complexity of the ECCS procedure and give a new parameterization of EC based on specification width.

First we recall the parameterization based on specification granularity [5].

Definition 14. *Let N_1, N_2 be two Boolean circuits with a CS S represented by partitions Spec(N_1)=$\{N_1^1,..,N_1^k\}$, and Spec(N_2)= $\{N_2^1,..,N_2^k\}$. The* ***granularity*** *of S is the size (i.e. the number of gates) of the largest subcircuit N_i^j, i=1,2, j=1,..,k.*

The ECCS procedure is exponential in the granularity p of S and linear in the number of subcircuits in *Spec(N_1)*, *Spec(N_2)* (i.e. in the number of mv-gates in S). The exponentiality in p is due to procedures *exist_quantify* and *correlation_function.* The reason why the ECCS procedure is exponential only in p and not in the circuit size is that the two exponential procedures above are applied only to subcircuits N_1^i, N_2^i whose size is bounded by p. So the time complexity of the ECCS procedure is the same as the size of a specification guided resolution proof from [5].

The granularity based parameterization can be easily improved using the following two observations. First, suppose that the number of outputs of a subcircuit N_i^j, i=1,2, j=1,..,k is bounded by a constant w. Then the complexity of the *correlation_function* procedure is bounded by 3^{4w}. Indeed, the number of variables of the function $D_{out}(N_1^i, N_2^i)$ is bounded by $2 * w$. Testing if $D_{out}(N_1^i, N_2^i)$ is a correlation function reduces to two SAT-checks over functions of $4 * w$ variables (see Remark 2) . In general resolution, the complexity of either check is bounded by 3^{4w} (the total number of clauses of $4 * w$ variables).

Second, to quantify from the function $H_i = \mathit{Sat}(v(N_1^i)) \wedge \mathit{Sat}(v(N_2^i)) \wedge D_{inp}(N_1^i, N_2^i)$ all the variables except the output variables of N_1^i, N_2^i, it suffices to perform no more than 2^{2w} SAT-checks. Each SAT-check just tests if a function $H_i \wedge A$ is satisfiable. Here A is a Boolean function that is equial to 1 iff a particular set of assignments to the outputs of N_1^i, N_2^i is made. So if the compelxity of each SAT-check is bounded by a function $f(w')$ where w' is another parameter, then the complexity of the ECCS procedure is bounded by $(f(w')*2^{2w}+3^{4w}) * k$ where k is the number of subcircuits in $\mathit{Spec}(N_1)$ and $\mathit{Spec}(N_2)$. So, if the values of w and w' are bounded, the complexity of the ECCS procedure is linear in the size of N_1 and N_2.

Let $\mathit{CNF}(H_i \wedge A)$ be a CNF representing $H_i \wedge A$. To parameterize the complexity of checking the satisfiability of $H_i \wedge A$ we use the results on FP tractability of SAT [8]. This FP tractability is achieved by parameterizing SAT with the width of a graph relating variables and clauses of a CNF formula. Given a formula F, one can build different graphs relating clauses and variables of F (e.g. incidence graph, primal graph, hypergraph [8]). Besides, one can use different definitions of graph width (e.g. cut-width [2, 3, 7], tree width, branch width [1, 8]). We do not choose a particular width-based parameterization here because the reasoning below is mostly independent of such a choice.

Denote by $G(H_i \wedge A)$ a graph of choice to describe a relation between clauses and variables of $\mathit{CNF}(H_i \wedge A)$. To complete parametrization we show how one can compute $G(H_i \wedge A)$. Of course its computing is trivial if $\mathit{CNF}(H_i \wedge A)$ is given explicitly. However, this is not the case. Nevertheless one can "approximate" $G(H_i \wedge A)$ as follows. $\mathit{CNF}(H_i \wedge A)$ can be represented as $\mathit{CNF}(\mathit{Sat}(v(N_1^i))) \wedge \mathit{CNF}(\mathit{Sat}(v(N_2^i))) \wedge \mathit{CNF}(D_{inp}(N_1^i, N_2^i)) \wedge \mathit{CNF}(A)$. Knowing circuits N_1^i and N_2^i one can easily build CNF formulas $\mathit{CNF}(\mathit{Sat}(v(N_1^i)))$, $\mathit{CNF}(\mathit{Sat}(v(N_2^i)))$. So their contribution to $G(H_i \wedge A)$ can be easily computed. $\mathit{CNF}(A)$ can be represented as a conjunction of unit clauses, each clause describing an assignment one has to make to an output of N_1 or N_2. Since the number of outputs of N_1^i and N_2^i is known, the contribution of $\mathit{CNF}(A)$ to $G(H_i \wedge A)$ is easy to compute. To compute the contribution of $\mathit{CNF}(D_{inp}(N_1^i, N_2^i))$ to $G(H_i \wedge A)$ we can just assume the "worst" case. Namely, if outputs of the circuit N_1^j (respectively N_2^j) are connected to inputs of N_1^i (respectively to inputs of N_2^i) we can assume that $\mathit{CNF}(D_{inp}(N_1^i, N_2^i))$ contains 3^q clauses each clause containing all the variables corresponding to the outputs of N_1^j and N_2^j. Here q is the total number of outputs of N_1^j and N_2^j. (By assumption, $q \leq 2 * w$.) The definition below summarizes our effort to parameterize EC by specification width.

Definition 15. *Let N_1 and N_2 be two Boolean circuits and partitions Spec(N_1)$=\{N_1^1,.., N_1^k\}$ and Spec(N_2)$=\{N_2^1,.., N_2^k\}$ represent their specifications. The* ***specification width*** *is the maximum of w and w' where w is the maximum number of outputs of a circuit N_i^j,i=1,2, j=1,., k and w' is the maximum width of graph $G(H_i \wedge A)$ for all i. (Graph $G(H_i \wedge A)$ is built as described above.)*

8 Why It Is Hard to Find a Common Specification

As we mentioned in Section 2 in contrast to an FP tractable problem, parameterized EC of circuits N_1, N_2 needs an "advice" in the form of a CS. A natural question is whether this advice is crucial or there is an efficient algorithm that, given circuits N_1, N_2, can efficiently find a CS of width w. In [5] it was conjectured that finding a CS is hard for any deterministic algorithm. (So EC of circuits may serve as a source of formulas that have linear complexity, say, in general resolution but are extremely hard for a deterministic algorithm.) Armed with the new definition of CS based on toggle equivalence we can better justify this conjecture.

Suppose that N_1 and N_2 are two circuits to be checked for equivalence and we only know that N_1 and N_2 have a CS of width w. To find such a CS one needs to find a partition $\{N_1^1,.., N_1^k\}$ of N_1 and a partition $\{N_2^1,\ldots, N_2^k\}$ of N_2 into subcircuits that are connected in the "same way" and are toggle equivalent. Here we face the following two big problems. The first problem is that finding a pair of subcircuits that are toggle equivalent is computationally very expensive. Even though the parameter w limits the number of outputs a subcircuit can have, the number of candidate subcircuits in N_j,j=1,2 is roughly speaking proportional to $|N_j|^w$ (here $|N_j|$ is the number of gates in N_j).

The second problem is that if N_1^i, N_2^i is a part of a CS S, they have to satisfy the following two conditions.

- One should be able to check toggle equivalence of N_1^i, N_2^i in terms of their local inputs restricted by previously computed constraint function D_{inp}.
- The same should hold for the subcircuits of S whose inputs are connected to outputs of N_1^i, N_2^i.

Thus, even if we find two subcircuits N_1^i, N_2^i of width w that are toggle equivalent, it does not necessarily mean that they are a part of any CS of N_1 and N_2 of width w. (It may be the case that they are toggle equivalent "accidentally" and so we will not be able to find any toggle equivalent subcircuits of width w that are connected to outputs of N_1^i and N_2^i.) Due to the two problems above, it is very unlikely that there is an efficient algorithm for finding a CS of bounded width.

9 Conclusions

We give an efficient procedure for checking the equivalence of Boolean circuits with a common specification of bounded width. This result implies that one can consider parameterized equivalence checking as a generalization of FP tractability of Circuit-SAT. Instead of looking for circuits that are easy in absolute terms (e.g circuits of bounded width) one can search for pairs of circuits that are easy "relatively" i.e. with respect to each other (e.g circuits with a CS of bounded width). One can view parameterized EC as a way to study "natural" transformations of a circuit i.e transformations that preserve its functionality and can be easily verified.

Acknowledgments

The author would like to thank Edward Hirsch of Steklov Institute of Mathematics at St.Petersburg, Felice Balarin of Cadence Berkeley Labs, and anonymous reviewers for their valuable feedback.

References

1. A. Alekhnovich, A.Razborov. *Satisfiability, Branch-width and Tseitin Tautologies.* FOCS-2002,pp. 593-603.
2. C.L.Berman. *Circuit width, Register Allocation and Ordered Binary Decision Diagrams.* IEEE Trans. CAD. 10(8),pp.1059-1066,Aug. 1991.
3. E. A. Broering, S. V. Lokam. *Width-Based Algorithms for Satisfiability.* Proceedings of SAT 2003. Lecture Notes in Computer Science (LNCS), Volume 2919, pp. 162-171, 2004.
4. R.G. Downey, M.R.Fellows. *Parameterized complexity.* Springer-Verlag 1999.
5. E.Goldberg,Y.Novikov. *How good can a resolution based SAT-solver be?* SAT-2003, LNCS 2919,pp.37-52.
6. E.Goldberg. *On equivalence checking and logic synthesis of circuits with a common specification.* Cadence Berkeley Labs, Technical report, CDNL-TR-2004-1220,August 2004, (http://eigold.tripod.com/papers/tr-2004-1220.pdf).
7. M.R.Prasad,P.Chong, and K. Keutzer. *Why is Combinatorial ATPG Efficiently Solvable for Practical VLSI Circuits?* Journal of Electronic Testing: Theory and Applications, vol. 17,pp.509-527,2001.
8. S. Szeider. *On Fixed-parameter Tractable Parameterizations of SAT.* Proceedings of SAT-2003. Lecture Notes in Computer Science (LNCS) Volume 2919, pp. 188-202, 2004.

Observed Lower Bounds for Random 3-SAT Phase Transition Density Using Linear Programming

Marijn Heule* and Hans van Maaren

Department of Software Technology,
Faculty of Electrical Engineering,
Mathematics and Computer Sciences,
Delft University of Technology
marijn@heule.nl
h.vanmaaren@ewi.tudelft.nl

Abstract. We introduce two incomplete polynomial time algorithms to solve satisfiability problems which both use Linear Programming (LP) techniques. First, the FLIPFLOP LP attempts to simulate a Quadratic Program which would solve the CNF at hand. Second, the WEIGHTEDLINEARAUTARKY LP is an extended variant of the LINEARAUTARKY LP as defined by Kullmann [6] and iteratively updates its weights to find autarkies in a given formula. Besides solving satisfiability problems, this LP could also be used to study the existence of autark assignments in formulas. Results within the experimental domain (up to 1000 variables) show a considerably sharper lower bound for the uniform random 3-SAT phase transition density than the proved lower bound of the myopic algorithm (> 3.26) by Achlioptas [1] and even than that of the greedy algorithm (> 3.52) proposed by Kaporis [5].

1 Introduction

Although Linear Programming (LP) techniques exist which run in polynomial time, it must be said that these techniques certainly are not among the most popular tools in the Satisfiability area. In those cases where they are used they are mainly applied in a preprocessing phase, in order to detect a specific structure of the CNF at hand. Examples of practical applications in this field using LP techniques are measuring the complexity of a CNF formula [2] and the detection of so-called equivalence clauses [9]. The reason why LP techniques are rarely used as a reasoning engine is that the straightforward LP relaxation of the Satisfiability Problem is always feasible (when unit clauses are absent). This means that in any trial to make LP techniques useful for CNF reasoning one has to come

* Supported by the Dutch Organization for Scientific Research (NWO) under grant 617.023.306.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 122–134, 2005.

up with a sophisticated reformulation. In this paper we introduce two LP based techniques which both result in a polynomial time running incomplete solver.

The first one, the FlipFlop LP, finds its motivation in the fact that the Satisfiability Problem can straightforwardly be reformulated as a Quadratic Programming Problem. We simulate this Quadratic Program through an iterated sequence of LP's, where the optimal solution of the n-th iteration is used as an input for the $n+1$-th iteration. That is, we reformulate the Quadratic Program as a "Contraction" problem, although not in the strict Mathematical sense. Satisfying assignments are among the "Contraction" fixed points, but unfortunately, the zero solution, which reveals no information at all, is also one of those fixed points. Hence, in order to make things work, we have to invoke certain tricks, which could be classified as "artificial", to stay away from this trivial fixed point.

The second LP is based on the detection of autarkies - partial assignments that satisfy all clauses that are "touched". We call it the WeightedLinearAutarky LP, which finds its origin in Kullmann's Linear Autarky detection [6]. Also in this case we introduce in fact a sequence of LP's in order to find suitable weights for autarky detection. Again, we have to invoke tricks and some tuning to obtain a satisfactory performance.

Both of the above methods are generally applicable, but here we want to focus on their performance on uniform random 3-Sat formulas. From the presented results it becomes evident that the second approach is monotonically better than the first on these set of instances. Nevertheless we present both techniques, since we observed that outside the domain of random 3-Sat the situation is sometimes reversed.

The search for lower bounds for the uniform random 3-Sat phase transition density has a rich history by now [3]. To our knowledge the best result found by myopic algorithms is due to Achlioptas [1], but the greedy algorithm of Kaporis *et al* [5] pushes this bound a bit further. We emphasize that this contribution does not claim to come up with an even sharper proven lower bound. In order to do so one needs algorithms which are subject to some profound (likely probabilistic) analysis and LP techniques do not easily subject themselves as such. At least, such an analysis is beyond the capacity of the authors at this stage.

The question we address here has a more modest objective: to what observed densities can polynomial time algorithms of a given complexity solve uniform random 3-Sat problems. The experimentally obtained results show that both of our LP based techniques have the potential to go well above Kaporis' bound. We find this interesting as such, but most of all we take these results as an indication that LP based techniques are useful tools in the Satisfiability area.

As far as presenting the capacity of our methods within the context of uniform random 3-Sat and relating our results to the lower bound question, one might object that local search based solvers easily almost reach the phase transition density and that because of this feature our methods are situated in a sort of a "twilight zone": no analysis possible at the moment and outperformed by local search methods. To some extent this is indeed the case, however the authors are not aware of the existence of a local search solver with a default scalable parameter setting resulting in an incomplete polynomial time solver reaching the phase transition density consistently. In case they exist, we hope to be alerted by the SAT community, and complexity comparisons must be made.

2 Preliminaries

A formula (denoted as $\mathcal{F} = c_1 \wedge c_2 \wedge \ldots \wedge c_m$) in *Conjunctive Normal Form* (CNF) is a conjunction of clauses; each clause (denoted as $c_i = l_1 \vee l_2 \vee \cdots \vee l_k$) being a disjunction of literals; and each literal is an atomic Boolean variable x_j or its negated form $\neg x_j$. The satisfiability (SAT) problem deals with the question whether a CNF formula is Satisfiable (has a Boolean solution) or not. A formula is satisfiable if an assignment satisfies all clauses. A clause is satisfied if at least one of its literals is satisfied.

The *clause-variable matrix* A associated to a CNF formula $\mathcal{F}$ is the matrix defined by

$$A_{i,j} = \begin{cases} 1 & \text{if the } i^{th} \text{ clause of } \mathcal{F} \text{ contains the } j^{th} \text{ variable with sign 1} \\ -1 & \text{if the } i^{th} \text{ clause of } \mathcal{F} \text{ contains the } j^{th} \text{ variable with sign -1} \\ 0 & \text{otherwise.} \end{cases}$$

Thus if $\mathcal{F}$ is a CNF formula with propositional variables $x_1, \ldots, x_n$ the SAT problem for $\mathcal{F}$ reads as the -1,1 Feasibility problem

$$\begin{cases} Ax \geq -L + 2e \\ \quad x \in \{-1, 1\}^n \end{cases}$$

In the above, L is the *length vector*, having the length of clause i as its i^{th} entry, and e is the all one vector of appropriate dimension. Notice that we use {-1,1} Boolean variables instead of the commonly used {0,1}.

As defined by Kullmann [6], a formula with clause-variable matrix A has a Linear Autarky $x \in \mathcal{Q}^n$ if

$$\begin{cases} Ax \geq 0 \\ \quad x \neq 0 \end{cases}$$

The above concept generalizes an earlier version of Warners and van Maaren [10] which provides a decomposition of a formula in case the kernel of its clause-variable matrix is non-zero.

A so-called *monotone variable* is the best known example of a Linear Autarky: if variable x_j appears only positive (negative) in the formula, vector $x = e_j$ $(x = -e_j)$, (e_j is the j^{th} unit vector), is a Linear Autarky. In general, a Linear Autarky x leads to an autark partial assignment of the formula involved by rounding. To examine this, let partial assignment sign be

$$\mathsf{sign}(x_j) = \begin{cases} 1 & \text{if } x_j > 0 \\ -1 & \text{if } x_j < 0 \\ \text{undefined} & \text{if } x_j = 0. \end{cases}$$

and substitute all defined $\mathsf{sign}(x_j)$ into the formula. If a clause i is affected by this substitution, that is, if a variable with defined sign occurs in clause i, this clause is obviously satisfied by the partial assignment since

$$\sum_j A_{ij}x_j = \sum_{x_j \neq 0} A_{ij}x_j \geq 0$$

and hence not all $A_{ij}x_j$ with defined $\mathsf{sign}(x_j)$ can be negative. The above implies that a Linear Autarky leads to a non-trivial decomposition of the formula at hand. One part is satisfied by sign and the other part contains only variables undefined by sign. Clearly, the latter part is Satisfiability Equivalent to the original formula: it is satisfiable if and only if the original formula is.

3 The FlipFlop LP

A Quadratic Program (QP) formulation of the SAT problem as defined in the preliminaries is shown in Fig. 1(a). Since max $\sum x_i^2$ is a quadratic objective function (not a linear), solving this QP could not be executed in polynomial time, unless P = NP. We propose a linear simulation of this QP that is incomplete - in contrast to the QP - but could solve CNF formulas in polynomial time. This linear simulation is shown in Fig. 1(b). We use parameters v_i that represent "guessed" outcomes of x_i in max $\sum x_i^2$.

$$\begin{array}{ll} \max & \sum x_i^2 \\ \text{s.t.} & \begin{cases} Ax \geq -e \\ -1 \leq x_i \leq 1 \end{cases} \end{array} \qquad\qquad \begin{array}{ll} \max & \sum v_i x_i \\ \text{s.t.} & \begin{cases} Ax \geq -e \\ -1 \leq x_i \leq 1 \end{cases} \end{array}$$

(a) (b)

Fig. 1. (a) A 3-SAT QP formulation; (b) A possible linear simulation of (a)

Solving difficult problems requires a close approximation of x_i by v_i. We experimented with initial weights $v_i := \sum_{x_i \in \mathcal{F}}$ - $\sum_{\neg x_i \in \mathcal{F}}$. This LP could solve uniform random 3-SAT instances up to density of approximately 2.5.

To stretch this result, we experimented with various update strategies for the weights v_i. One rather intuitive update strategy applies the outcomes of x_i in the solution of max $\sum v_i x_i$ as an "educated guess" for the weights in a new iteration: $v_i^{n+1} := \mathrm{R}^n(x_i)$ with $\mathrm{R}^n(x_i)$ referring to the value of x_i in the solution of the linear simulation after iteration n. Even if one allows many iterations, this LP is unable to solve many problems apart from the ones that could be solved using the initial weights.

Of the various update strategies we used during our experiments, one clearly solved the most problems. The recipe: divide the variables in two disjunct sets $\mathcal{A}$ and $\mathcal{B}$. Variables were randomly divided between sets $\mathcal{A}$ and $\mathcal{B}$ with an equal

$$x_i \in \begin{cases} \mathcal{A} & p(0.5) \\ \mathcal{B} & \text{otherwise} \end{cases} \qquad v_i^{n+1} := \begin{cases} \mathrm{R}^n(x_i) & \text{if } x_i \in \mathcal{A} \textbf{ and } n \equiv 0 \ (mod\ 2) \textbf{ or} \\ & \text{if } x_i \in \mathcal{B} \textbf{ and } n \equiv 1 \ (mod\ 2) \\ 0 & \text{otherwise} \end{cases}$$

(a) (b)

Fig. 2. (a) Initial set definition; (b) Update strategy

probability. The weights are updated in such a way that the weights become 0 if the corresponding variable is alternately in set $\mathcal{A}$ or set $\mathcal{B}$. This update strategy attempts to assign variables - as many as possible - in one set to the sign of the corresponding weights. This results in assigning values $\neq 0$ to variables in the other set. The $\mathrm{R}^n(x_i)$ values will be used in iteration $n + 1$ where the same process is performed, but with swapped sets (see Fig. 2). We refer to iteratively solving the linear simulation defined in Fig. 1(b) using the update strategy as defined in Fig. 2 as the FlipFlop LP.

4 Weighted Linear Autarkies

In this section, we explain the concept of *weighted linear autarkies* and the corresponding LP. A proper weight matrix is required to solve CNF's. First we show how LP techniques can be used to determine the importance of the variables. Second, we propose an update strategy which use this information to construct the weight matrix.

4.1 The Weighted Linear Autarky LP

We refer to the *weighted clause-variable matrix* W associated to a CNF formula $\mathcal{F}$ as the weighted analogue of matrix A using weights $w_{i,j} > 0$:

$$W_{i,j} = \begin{cases} w_{i,j} & \text{if the } i^{th} \text{ clause of } \mathcal{F} \text{ contains the } j^{th} \text{ variable with sign 1} \\ -w_{i,j} & \text{if the } i^{th} \text{ clause of } \mathcal{F} \text{ contains the } j^{th} \text{ variable with sign -1} \\ 0 & \text{otherwise.} \end{cases}$$

The definition of the LinearAutarky LP as proposed by Kullmann [6] is shown in Fig. 3(a). Our variant uses the weighted clause-variable matrix W (see Fig. 3(b)). We refer to this LP as the WeightedLinearAutarky LP. For both LP's, one can try to achieve the constraint $x \neq 0$ by an objective function. Two possible objective functions are shown in Fig. 4.

$$\begin{cases} Ax \geq 0 \\ \quad x \neq 0 \end{cases} \qquad \begin{cases} Wx \geq 0 \\ \quad x \neq 0 \end{cases}$$

(a) (b)

Fig. 3. (a) LinearAutarky LP; (b) WeightedLinearAutarky LP

$$\max \quad \sum x_i$$

$$(a)$$

$$\begin{aligned} \max \quad & \sum v_i x_i \\ & v_i := \textstyle\sum_{x_i \in \mathcal{F}} - \sum_{\neg x_i \in \mathcal{F}} \end{aligned}$$

$$(b)$$

Fig. 4. (a) Elementary objective function; (b) Advanced objective function

The advanced objective function seemed more effective, since it finds all monotone variables directly. This in contrast to the elementary objective function which will not find monotone variables that occur only negative. However, during our experiments we found no differences in the solving capacity of both objective functions. Therefore we selected the elementary one.

As with the LinearAutarky LP, the set of variables with $x_i \neq 0$ in the solution of a WeightedLinearAutarky LP forms an autark assignment. The advantage of the WeightedLinearAutarky LP is that it could - in theory - solve many problems. The following theorem is quite similar to Lemma 3.2 by Kullmann in [7].

Theorem 1. *For every satisfiable formula $\mathcal{F}$, there exists a weighted clause-variable matrix W of which its* WeightedLinearAutarky *LP results in a solution.*

Proof: Given a satisfying assignment of $\mathcal{F}$, construct W in such a way that satisfied literals $x_{i,j}$ have corresponding weight $w_{i,j} = L_i - 1$ and falsified literals $x_{i,j}$ have corresponding weight $w_{i,j} = 1$. L_i refers to the length of clause i. The WeightedLinearAutarky LP will result in the given satisfying assignment. □

In practice, it is hard to construct a W leading to a solution. We propose a method to construct matrix W using Linear Programming techniques. First, we solve a LP called the UpdateWeights LP, which is quite similar to the WeightedLinearAutarky LP. Second, we use the values of variables x_i in the solution of that UpdateWeights LP to construct W by applying an *update strategy*.

4.2 The Update Weights LP

We saw that solving the LP with constraints $Wx \geq 0$ and $x \neq 0$ would result in an autark assignment if it is feasible. The LP with constraints $Wx + y \geq 0$, $x \neq 0$, and $y > 0$ is always feasible, but would rarely result in an autark assignment. However, solutions for this LP could be useful to update W in such a way that the LP with constraints $Wx \geq 0$ and $x \neq 0$ becomes feasible. This idea is the foundation of the UpdateWeights LP as defined in figure 5.

The values of parameters y_{MIN} and y_{MAX} are essential for the solving capacity of the algorithm. Since the variables x_i are unbounded, there is only one parameter that needs to be measured: $\frac{y_{\text{MAX}}}{-y_{\text{MIN}}}$. One must choose y_{MIN} and y_{MAX} in such a way

$$\max \quad \sum y_i \qquad\qquad \begin{cases} Wx + y \geq 0 \\ x \neq 0 \\ y_{\text{MIN}} \leq y_i \leq y_{\text{MAX}} \end{cases}$$

$$(a) \qquad\qquad\qquad (b)$$

Fig. 5. UpdateWeights LP definitions: (a) objective function; (b) constraints

that in as many relaxations of clauses at least one $x_{i,j}$ exists with $\mathsf{sign}(x_{i,j}) = 1$. If $\frac{y_{\text{MAX}}}{-y_{\text{MIN}}}$ is too large, many clauses will have either three literals with $\mathsf{sign}(x_{i,j}) = 1$ or three literals with $\mathsf{sign}(x_{i,j}) = -1$. On the other hand, if $\frac{y_{\text{MAX}}}{-y_{\text{MIN}}}$ is too small, many variables will have an undefined $\mathsf{sign}(x_{i,j})$. Throughout our experiments on uniform random 3-Sat formulas, $\frac{y_{\text{MAX}}}{-y_{\text{MIN}}} \approx 2$ appeared to solve most instances near the phase transition density. More specific, we used y_{MIN} := -0.3 and y_{MAX} := 0.6.

4.3 Update Strategy

We refer to $\mathrm{S}^n(x_i)$ as the value of x_i in the solution of the UpdateWeights LP after iteration n. We want to use these values $\mathrm{S}^n(x_i)$ to construct matrix W^{n+1}. An update strategy must satisfy the definition property that all $w_{i,j} > 0$. Notice that $\mathrm{S}^n(x_i)$ is unbounded, since variables x_i are unbounded in the UpdateWeights LP. Now it seems natural to make $w_{i,j}^{n+1} := base^{S^n(x_i)w_{i,j}^n}$. To prevent that weights might reach either 0 or ∞, we introduce two parameters w_{MIN} and w_{MAX} which refer to the minimum and the maximum of the weights, respectively. This results in the following strategy:

$$\begin{cases} W^0 = A \\ w_{i,j}^{n+1} = base^{S^n(x_i)w_{i,j}^n} \\ 0 < w_{\text{MIN}} \leq w_{i,j} \leq w_{\text{MAX}} \end{cases}$$

Fig. 6. Update strategy for the WeightedLinearAutarky LP

Parameter w_{MIN} has no significant influence on the performance of the algorithm as long as its value is small. We chose w_{MIN} := 0.01. A small performance gain could be achieved by making $w_{\text{MAX}} \approx 5$ (assuming y_{MIN} = -0.3 and y_{MAX} = 0.6). We did some small scale experiments to determine an effective value for parameter *base*: using the other parameters on their chosen values as mentioned above, we experimented with uniform random 3-Sat formulas. We generated 1000 instances on density 3.7 using 200 variables. The influence of parameter *base* on the performance was measured starting with *base* = 1 using steps of 0.25 and from 4.0 we used only integer values.

Results are shown in Fig. 7. The optimal value *base* = 2.5 shown here is used during our further experiments. Notice that for *base* = 1 the WeightedLinearAutarky LP equals the Linear Autarky LP. So using the LinearAutarky LP, we cannot solve one single sample of these instances.

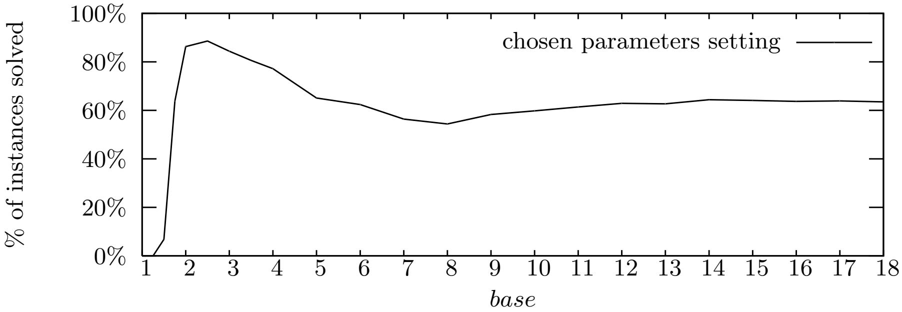


Fig. 7. Influence of parameter *base* on the solving capacity of the algorithm

Although intensively tweaking of the various parameters might result in a better performance of the algorithm, we used these settings for our experiments.

5 Experimental Results

This section deals with the performance of the two solvers based on the LP's described above. Our goal is to observe a lower bound for the uniform random 3-Sat phase transition density. During the experiments we used an Intel Pentium 4 with 3.5 GHz running on Fedora Core 2. All LP's were solved using glpsol from the GNU Linear Programming Kit[1] (GLPK). To generate the uniform random 3-Sat formulas, we used the generator of Van Gelder[2] [8].

As we will see, the WeightedLinearAutarky LP clearly performs better than the FlipFlop LP on uniform random 3-Sat formulas. The reason for presenting the results of the latter is the fact that for some structured problems we observed a reverse effect. Presentation of these results is outside the scope of this paper.

Since Linear Programming is a polynomial time solving procedure, the complexity of the presented LP's is also polynomial. But only, of course, as long as the required iterations to solve a formula does not grow exponentially with respect to its size. We used a small constant to limit the number of iterations (MAX_ITERATIONS := 50). Using this constant, we already reached a point where, as the number of variables increased, the density on which all instances were solved increased accordingly. A higher constant can only increase the performance of the solvers using the presented LP's. This will be explained later. For reasons of comparison we added the results of Kaporis' algorithm on the same samples (see Fig. 12) at the end of this section.

[1] available at `http://www.gnu.org/software/glpk/glpk.html`

[2] available at `http://www.satlib.org`

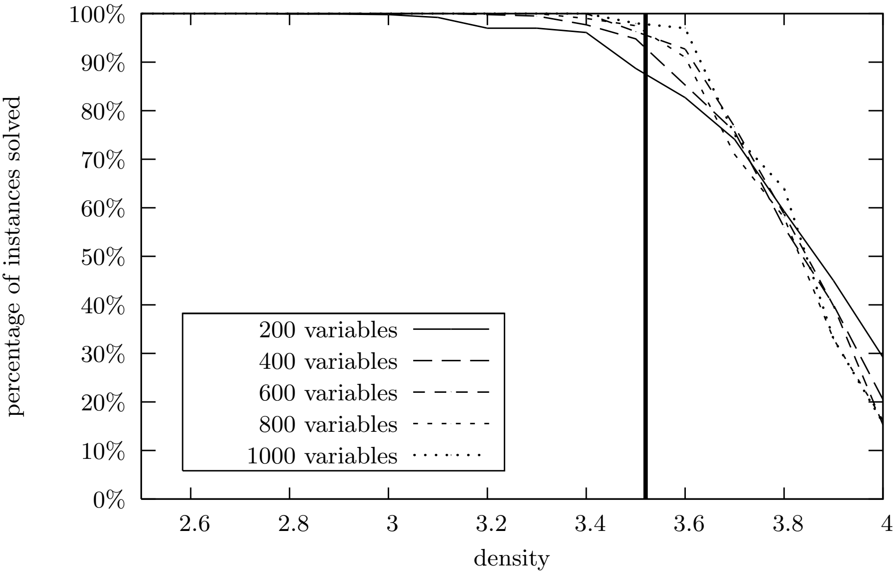


Fig. 8. Percentage of instances solved using the FlipFlop LP. Vertical line shows the proved lower bound by Kaporis [5]

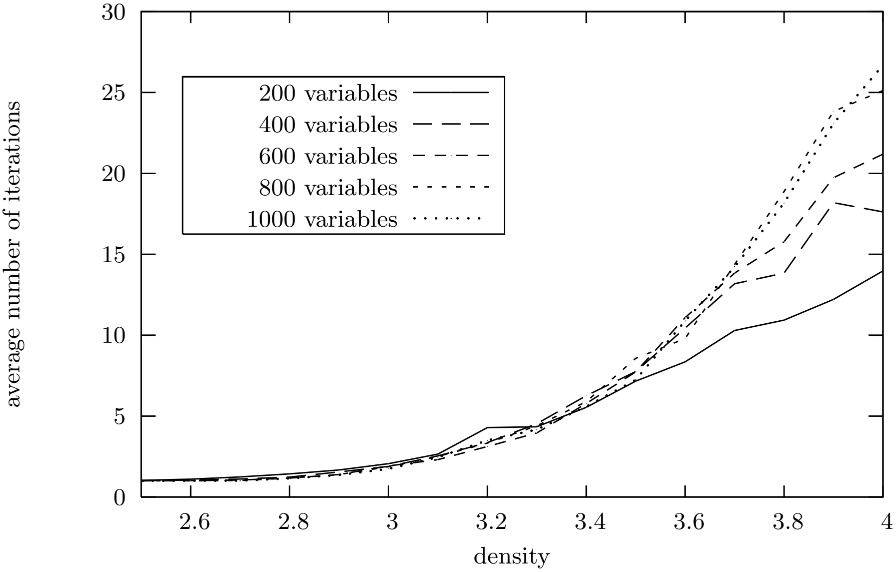


Fig. 9. Average number of iterations required by the FlipFlop LP

5.1 Flip Flop Solver

The FlipFlop solver (see algorithm 1) first attempts to assign all variables to {-1,1} by using initial weights based on the occurrences of the variables in

$\mathcal{F}$. In the first iteration these weights are used to solve the linear simulation in Fig. 1(b). For all next iterations, it uses the FlipFlop update strategy. A solution is found when all variables are assigned a value {-1,1} by the LP.

Algorithm 1 FlipFlop solver($\mathcal{F}$)

```
for $i \in \{1,..,n\}$ do
    $v_i := \sum_{x_i \in \mathcal{F}} - \sum_{\neg x_i \in \mathcal{F}}$
end for
for $j \in \{1,.., \text{MAX\_ITERATIONS}\}$ do
    $v$ := FlipFlopLP($A_{\mathcal{F}}$, $v$, $j$ modulo 2)
    if $\sum_{i=1}^{n} v_i^2 = n$ then
        return SATISFIABLE
    end if
end for
return UNKNOWN
```

We experimented for each density (using steps of 0.1) with 1000 instances of 200 variables, 600 instances of 400 variables, 400 instances of 600 variables, 200 instances of 800 variables and 100 instances of 1000 variables. Figure 8 shows the percentage of instances solved by the FlipFlop solver. We observed that by increasing the number of variables, higher densities are reached where all instances were solved. However, it consequently lags behind the WeightedLinearAutarky solver.

Figure 9 shows the average number of iterations required to find a solution of the solved instances. Recall that we terminate the solver after 50 iterations. Up to density 3.0 these averages are slightly lower than the averages by the WeightedLinearAutarky solver (see for comparison Fig. 11). The FlipFlop solver owes its performance to the many instances that were solved in the first iteration - due to the initial weights.

5.2 Weighted Linear Autarky Solver

The WeightedLinearAutarkySolver, as shown in algorithm 2, attempts to solve a formula $\mathcal{F}$ by alternately solving the WeightedLinearAutarky LP and the UpdateWeights LP. We used the update strategy as defined above to construct W in each iteration.

The WeightedLinearAutarkyLP(W$_{\mathcal{F}}$) returns set $\mathcal{I}$, which contains all variables with $x_i \neq 0$. If $\mathcal{I} \neq \emptyset$, then the LP found an autark assignment. All clauses that contain at least one variable in $\mathcal{I}$ are removed from the formula. This is denoted by IterativeUnitPropagation($\mathcal{F} \cap \mathcal{I}$). If no clause is left in $\mathcal{F}$ after this removal, the formula is satisfiable.

As with the FlipFlop solver, we experimented for each density with 1000 instances of 200 variables, 600 instances of 400 variables, 400 instances of 600 variables, 200 instances of 800 variables and 100 instances of 1000 variables (using steps of 0.1). Figure 10 shows the percentage of instances solved by the

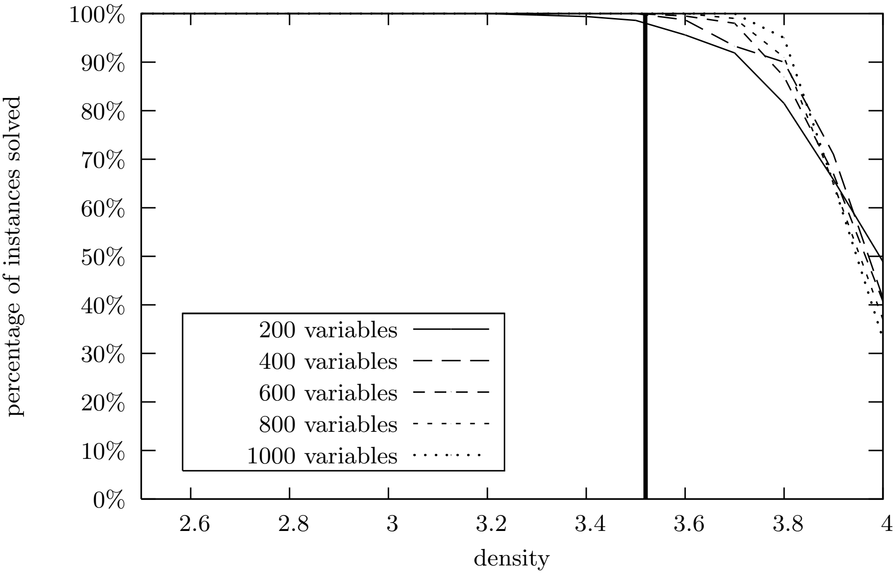


Fig. 10. Percentage of instances solved using the WeightedLinearAutarky LP. Vertical line shows the proved lower bound by Kaporis [5]

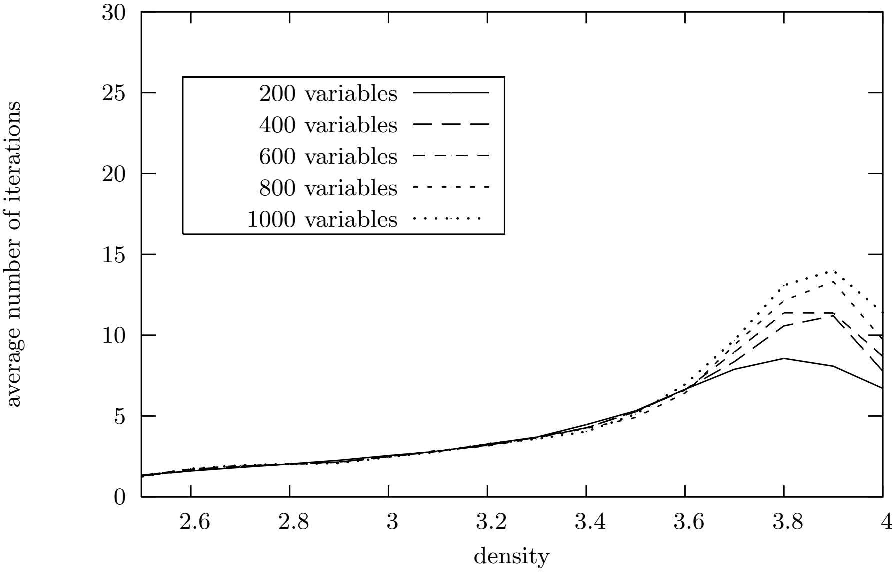


Fig. 11. Average number of iterations required by the WeightedLinearAutarky LP

Algorithm 2 WeightedLinearAutarkySolver($\mathcal{F}$)

```
$W_{\mathcal{F}} := A_{\mathcal{F}}$
for $j \in \{1, .., \text{MAX\_ITERATIONS}\}$ do
  $\mathcal{I}$ := WeightedLinearAutarkyLP($W_{\mathcal{F}}$)
  if $\mathcal{I} \neq \emptyset$ then
    $\mathcal{F}$ := IterativeUnitPropagation($\mathcal{F} \cap \mathcal{I}$)
    $W_{\mathcal{F}} := A_{\mathcal{F}}$
    if $\mathcal{F} = \emptyset$ then
      return SATISFIABLE
    end if
  else
    $W_{\mathcal{F}}$ := UpdateWeightsLP($W_{\mathcal{F}}$)
  end if
end for
return UNKNOWN
```

WeightedLinearAutarky solver. With an increasing number of variables come higher densities where all instances were solved: all instances consisting of 200 variables on density 3.4 were solved, while on density 3.7 all instances consisting of 1000 variables were solved. Figure 11 shows the average number of iterations required to find a solution of the solved instances. Although there is a small increase in this average - while increasing the number of variables - the number of iterations divided by the number of variables is decreasing.

During our experiments, a vast majority of the found autark assignments were either monotone variables or satisfying assignments. Up to density 3.4, we found some autark assignments consisting of multiple literals that only satisfied a part of the formula. However, this only occurred in approximately 0.5% of the formulas, regardless of the number of variables.

6 Conclusions

Within the experimental domain, both our methods show a threshold shaped success performance ratio of proving Satisfiability with respect to increasing density, growing steeper with increasing size. In this context, our second method shows a better performance. If the reader allows us to extrapolate to infinity the observed success ratio of 100% seems to be located well above the Kaporis' bound. For reasons of comparison we implemented the greedy algorithm of Kaporis described in [4] and investigated its performance on the same samples. The emerged figure 12 shows that this algorithm has to catch up quite a bit within the range $[1000, \infty]$. From its decreasing performance with growing size (within our domain of experimentation) above the Kaporis' bound one could guess that for very large size instances this success performance ratio must have a perfect threshold shape. Our main conclusion however is that LP techniques, having a modest polynomial complexity, apparently are able to reveal hidden structure in CNF's, even in case of randomly generated formulas. The interesting aspects lie in the fact that they are not evidently resolution

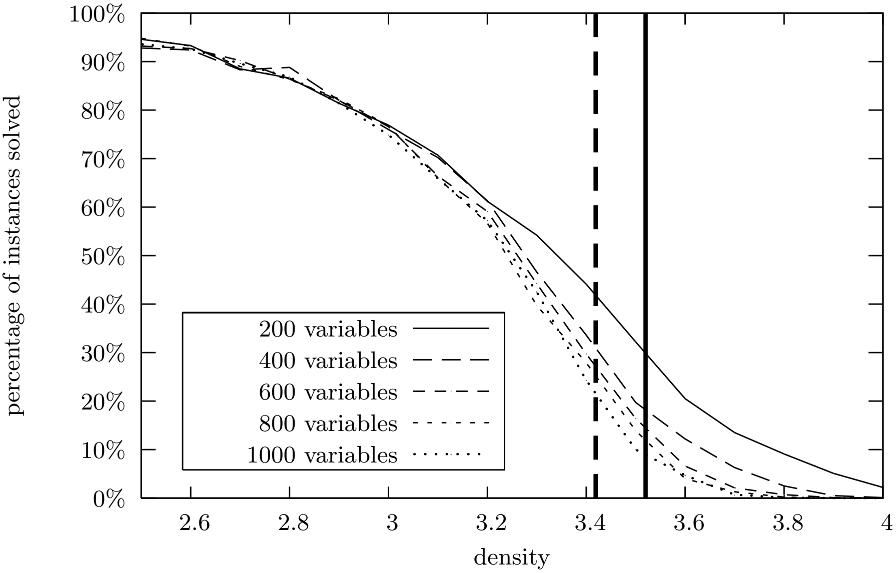


Fig. 12. Performance of the greedy algorithm proposed by Kaporis [4]. Dashed vertical line shows the proved lower bound by Kaporis in [4]; straight vertical line shows the proved lower bound by Kaporis in [5]

based and that their reasoning mechanisms, which clearly must be there, are rather unconventional compared to the standard ones used in the Satisfiability area.

References

1. D. Achlioptas, *Lower Bounds for Random 3-SAT via Differential Equations.* Theoretical Computer Science **265**(1-2) (2001), 159–185.
2. E. Boros, Y. Crama, P. Hammer, and M.Saks, *A complexity Index for Satisfiability Problems.* SIAM Journal on Computing **23** 1 (1994), 45–49
3. J. Franco, *Results related to threshold phenomena research in Satisfiability: lower bounds.* Theoretical Computer Science **265**(1-2) (2001), 147–157.
4. A.C. Kaporis, L.M. Kirousis, E.G. Lalas, *The Probabilistic Analysis of a Greedy Satisfiability Algorithm.* Lecture Notes In Computer Science **2461** (2002), 574–585.
5. A.C. Kaporis, L.M. Kirousis, E.G. Lalas, *Selecting complementary pairs of literals.* Electronic Notes in Discrete Mathematics **16** (2003).
6. O. Kullmann, *Investigations on autark assignments.* Discrete Applied Mathematics **107**(1-3) (2000), 99–137.
7. O. Kullmann, *Lean clause-sets: generalizations of minimally unsatisfiable clause-sets.* Discrete Applied Mathematics **130**(2) (2003), 209–249.
8. A. Van Gelder, Problem generator `mkcnf.c` contributed to the DIMACS Challenge archieve.
9. J.P. Warners, H. van Maaren, *A two phase algorithm for solving a class of hard satisfiability problems.* Oper. Res. Lett. **23**(3-5) (1998), 81-88.
10. J. Warners and H. van Maaren, *Solving satisfiability problems using elliptic approximations. Effective branching rules.* Discrete Applied Mathematics **107**(1-3) (2000), 241–259.

Simulating Cutting Plane Proofs with Restricted Degree of Falsity by Resolution

Edward A. Hirsch[1,⋆] and Sergey I. Nikolenko[2]

[1] St.Petersburg Department of Steklov Institute of Mathematics,
27 Fontanka, 191023 St.Petersburg, Russia
http://logic.pdmi.ras.ru/~hirsch/

[2] St.Petersburg State University
http://logic.pdmi.ras.ru/~sergey/

Abstract. Goerdt [Goe91] considered a weakened version of the Cutting Plane proof system with a restriction on the degree of falsity of intermediate inequalities. (The degree of falsity of an inequality written in the form $\sum a_i x_i + \sum b_i(1 - x_i) \geq A,\ a_i, b_i \geq 0$ is its constant term A.) He proved a superpolynomial lower bound on the proof length of Tseitin-Urquhart tautologies when the degree of falsity is bounded by $\frac{n}{\log^2 n+1}$ (n is the number of variables).

In this paper we show that if the degree of falsity of a Cutting Planes proof Π is bounded by $d(n) \leq n/2$, this proof can be easily transformed into a resolution proof of length at most $|\Pi| \cdot \binom{n}{d(n)-1} 64^{d(n)}$. Therefore, an exponential bound on the proof length of Tseitin-Urquhart tautologies in this system for $d(n) \leq cn$ for an appropriate constant $c > 0$ follows immediately from Urquhart's lower bound for resolution proofs [Urq87].

1 Introduction

During the past forty years the research concerning propositional proof systems was advancing mostly by proving exponential lower bounds on the length of proofs of specific tautologies in specific systems. For example, the resolution proof system had been a subject of a very thorough study that yielded exponential lower bounds for the propositional pigeonhole principle [Hak85], Tseitin-Urquhart tautologies [Tse68, Urq87], random formulas [BSW01] and many other families of tautologies. Also algebraic proof systems (the ones that deal with polynomial equalities) attracted a lot of attention in the 90s and similar lower bounds were proved for them (see [BGIP01] and references therein).

Much less is known about semialgebraic proof systems, which restate a Boolean tautology as a system of inequalities and prove that this system has no

[⋆] Supported in part by *Russian Science Support Foundation*, by *INTAS* grant #04-77-7173, and by grant #NSh-2203-2003-1 of the *President of RF* for leading scientific schools support.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 135–142, 2005.

0/1-solutions. No exponential lower bounds are known for higher degree proof systems (though there are exponential bounds for systems of inequalities that are *not* produced from Boolean tautologies; see [GHP02] and definitions and references therein). Concerning proof systems that work with linear inequalities, exponential lower bounds are known only for variations of the clique-coloring principle (see [Pud97] for a bound for the Cutting Plane proof system and [Das02] for a bound for a restricted version of the Lovasz-Schrijver system [LS91]).

The technique that allows to prove the bounds for linear semialgebraic proofs uses monotone interpolation and Razborov's lower bound on the monotone complexity of the clique problem [Raz85]. Therefore it is not suitable for proving lower bounds, for example, for Tseitin-Urquhart tautologies. Goerdt [Goe91] introduced a weaker version of Cutting Plane proofs by restricting the *degree of falsity* of inequalities (see Definition 1 below). He proved a superpolynomial lower bound on the proof length of Tseitin-Urquhart tautologies in the restricted system when the degree of falsity is bounded by $\frac{n}{\log^2 n+1}$, n being the number of variables. Goerdt's proof is a purely combinatorial argument obtained by modifying Urquhart's proof for resolution [Urq87].

In this paper we show the relation between the Cutting Plane proofs with restricted degree of falsity and the resolution proofs. Namely, we show that a length l Cutting Plane proof with degree of falsity bounded by $d(n) \leq n/2$ can be easily transformed into a resolution proof of length at most $l \cdot \binom{n}{d(n)-1} 64^{d(n)}$. Therefore, an exponential bound on the proof length of Tseitin-Urquhart tautologies for $d(n) \leq cn$ for an appropriate constant c follows immediately from Urquhart's lower bound [Urq87] on the length of resolution proofs.

2 Definitions

2.1 Propositional Proof Systems and Formulas

A *proof system* [CR79] for a language L is a polynomial-time computable function mapping strings in some finite alphabet (proof candidates) onto L (whose elements are considered as theorems). In this paper we are interested in a specific (yet very important) kind of proof systems: proof systems for the co-NP-complete language of unsatisfiable formulas in CNF (equivalently, tautologies in DNF).

In what follows, we use x to denote a variable (if not otherwise stated), l to denote a literal (i.e., a variable x or the negation $\overline{x}$ of it), and a, b, c, d, e, A, B to denote non-negative integers (all these letters may bear subscripts). A formula in CNF is a set of clauses, which are disjunctions (i.e., again sets) of literals and are usually denoted by letters C, D. We assume that a clause cannot contain a variable together with its negation.

A truth assignment π for a set of variables assigns a value (either 0 or 1) to each variable; the result of substituting π into a formula F is denoted $F|_\pi$ (where the clauses containing satisfied literals are removed, and falsified literals are dropped from the remaining clauses). We define the result of substituting π into other objects (clauses, inequalities, etc.) by analogy.

The proof systems we consider are dag-like derivation systems, i.e., a proof is a sequence of *lines* such that every line is either an axiom or is obtained by an application of a derivation rule to several previous lines. The proof finishes with a line called *goal* (in our case, this will be a simple form of contradiction). Such a proof system is thus determined by its set of axioms, set of derivation rules and the notion of a goal. Note that different proof system may have different notions of a line (it may be a clause, or an inequality, or anything else).

2.2 Resolution

The resolution proof system [Rob68] has clauses as its proof lines. Given a formula in CNF, one takes its clauses as the axioms and uses two rules:

- Resolution:
$$\frac{C \cup l \qquad D \cup \bar{l}}{C \cup D}$$
provided there is no literal $l' \in C$ such that $\overline{l'} \in D$.
- Weakening:
$$\frac{C}{C \cup l}$$
provided $\bar{l} \notin C$.

The goal is to derive the empty clause.

2.3 The Cutting Plane Proof System

The proof lines in the Cutting Plane proof system (**CP**) are linear inequalities with integer coefficients. To refute a formula in CNF in this system, one translates each its clause $l_1 \vee \ldots \vee l_k$ into the inequality $l_1 + \ldots + l_k \geq 1$, where a negative literal $l_i = \neg x_i$ is written as $1 - x_i$; the obtained system of linear inequalities has the same 0/1-solutions as the original set of clauses (where 1 corresponds to True, and 0 corresponds to False). The proof lines are *algebraic*, i.e., when we write $x + (1 - y) \geq 1$, we mean $x - y \geq 0$. **CP** allows to derive a contradiction (i.e., the inequality $0 \geq 1$) if and only if the original set of inequalities has no 0/1-solutions.

We state the initial inequalities as axioms, and add also the axioms

$$\frac{}{x \geq 0}, \qquad \frac{}{1 - x \geq 0} \tag{1}$$

for every variable x. The derivation rules are (x_i's denote variables, i ranges over all variables subscripts, other letters denote integer constants):

- Addition:
$$\frac{\sum a_i x_i \geq A \qquad \sum b_i x_i \geq B}{\sum (a_i + b_i) x_i \geq A + B}. \tag{2}$$

- Rounding rule:
$$\frac{\sum a_i x_i \geq A}{\sum \frac{a_i}{c} x_i \geq \lceil \frac{A}{c} \rceil}, \tag{3}$$
provided $c \geq 1$ and $\forall i \quad c | a_i$.
- Multiplication rule:
$$\frac{\sum a_i x_i \geq A}{\sum c a_i x_i \geq cA}, \tag{4}$$
where $c \geq 1$.

The notion of the degree of falsity of an inequality was introduced by Goerdt in [Goe91] as follows:

Definition 1. *Consider an inequality ι of the form $\sum_{i=1}^{n} a_i x_i \geq A$, and let π be a truth assignment for the variables of ι. The* degree of falsity *of ι under π is given by*
$$\mathtt{DGF}(\iota, \pi) = A - \mathtt{LHS}(\iota, \pi),$$
where $\mathtt{LHS}(\iota, \pi)$ is the value of the left-hand side of ι under π.

The degree of falsity of ι is given by
$$\mathtt{DGF}(\iota) = \max_{\pi} \ \mathtt{DGF}(\iota, \pi).$$

The degree of falsity of a Cutting Plane proof Π is given by
$$\mathtt{DGF}(\Pi) = \max_{\iota \in \Pi} \ \mathtt{DGF}(\iota).$$

3 Boolean Representations of Inequalities

Definition 2. *The* literal form *of an inequality is obtained from its canonical form $\sum_i c_i x_i \leq A$ by replacing each summand $c_i x_i$ where $c_i < 0$ with $-c_i(1 - x_i)$ and adding the corresponding constant $-c$ to the free coefficient. (The terms of the form x_i or $(1 - x_i)$ are called* literals*.)*

Example 1. For example, $2(1 - x) + 2y + (1 - z) \geq 3$ is the literal form of $-2x + 2y - z \geq 0$.

Lemma 1. $\mathtt{DGF}(\iota)$ *is the free coefficient of ι written in the literal form.*

Proof. Note that minimizing $\mathtt{LHS}(\iota, \pi)$ in the definition of $\mathtt{DGF}(\iota)$ is a trivial task: we should assign 0 to each variable x_i such that $a_i > 0$, and assign 1 to each x_i such that $a_i < 0$ (in terms of Definition 1). This assignment π_0 yields $\mathtt{DGF}(\iota, \pi_0) = A - \sum_{i: a_i < 0} a_i$, which is equal to the free coefficient of ι in the literal form. □

Lemma 2. *An integer inequality ι with $\texttt{DGF}(\iota) \le \frac{n}{2}$ can be represented as a conjunction of at most $\binom{n}{\texttt{DGF}(\iota)-1}$ Boolean clauses, where n is the number of variables in it.*

Proof. Consider all inequalities obtained by satisfying literals occurring in the literal form of ι with sum of coefficients up to $\texttt{DGF}(\iota)-1$. That is, we satisfy literals one by one and stop just before the inequality trivializes (i.e., the degree of falsity becomes non-positive), whatever coefficient we would choose next; we consider all inequalities that can be obtained from ι in this way (dropping duplicates, of course). It is easy to see that the obtained inequalities are equivalent to Boolean clauses. Indeed, consider an inequality $\sum_1^n a_i l_i \ge c$, where $\forall i \; a_i \ge c$. This inequality holds iff any one of l_i is true, which is equivalent to $l_1 \vee l_2 \vee \ldots \vee l_n$.

The number of clauses is as claimed, because we cannot satisfy more than $\texttt{DGF}(\iota) - 1$ literals without making ι trivial, and if an assignment results in a clause, its sub-assignments don't.

We have so far established a set of clauses that follows from the initial inequality. To prove the converse, consider an assignment π that falsifies ι. Substitute its part that satisfies literals of (the literal representation of) ι. The obtained inequality $\sum_{j\in J} a_j l_j \ge c' > 0$ is still non-trivial, because the original assignment falsifies ι. Then continue satisfying the remaining l_j's similarly to the construction above until the inequality becomes a clause. Clearly, this clause is falsified by (the remaining part of) π. □

Definition 3. *We call the set of clauses obtained from an inequality ι by the procedure described in the proof of Lemma 2 the* Boolean representation *of an inequality ι and denote it by $\mathcal{B}(\iota)$.*

Lemma 3. *If ι is derived from $\{\iota_j\}_{j\in S}$ in* **CP** *then, for each $C \in \mathcal{B}(\iota)$, there is a resolution proof of C from $\bigcup_{j\in S} \mathcal{B}(\iota_j)$ that only contains literals occurring in $\{C\} \cup \bigcup_{j\in S} \mathcal{B}(\iota_j)$.*

Proof. By Lemma 2, ι and $\mathcal{B}(\iota)$ have the same set of 0/1 solutions. Since the Cutting Plane proof system is sound and the resolution proof system is implicationally complete, the lemma follows (it is easy to see that one can get rid of the literals that do not occur in $\{C\} \cup \bigcup_{j\in S} \mathcal{B}(\iota_j)$: it suffices to eliminate the applications of the weakening rule introducing such literals). □

4 Simulation by Resolution

In this section we prove the bounds by establishing a direct connection between proofs in **CP** and proofs in the resolution proof system.

Lemma 4. *The rounding and multiplication rules do not change the Boolean representation.*

Proof. Suppose that ι' is obtained from ι by the rounding rule

$$\frac{\iota: \quad \sum_{i\in I} c_i l_i \geq A}{\iota': \quad \sum_{i\in I} \frac{c_i}{c} l_i \geq \left\lceil \frac{A}{c} \right\rceil},$$

where $c|c_i$ for all $i \in I$. For each clause $C \in \mathcal{B}(\iota)$, there is a (partial) assignment π that produced C from ι:

$$\iota|_\pi : \quad \sum_{i\in J} c_i l_i \geq A - \sum_{i\in I\setminus J} c_i.$$

Substitute this assignment into ι':

$$\iota'|_\pi : \quad \sum_{i\in J} \frac{c_i}{c} l_i \geq \left\lceil \frac{A}{c} \right\rceil - \sum_{i\in I\setminus J} \frac{c_i}{c}.$$

Then $\iota'|_\pi$ is also equivalent to a clause, because $\lceil \frac{A}{c} \rceil - \sum_{i\in I\setminus J} \frac{c_i}{c} \geq \frac{1}{c} \cdot (A - \sum_{i\in I\setminus J} c_i) > 0$ and (since $c|c_k$ for all k) $\forall j \in J$

$$\begin{aligned}\frac{c_j}{c} - \left(\left\lceil \frac{A}{c} \right\rceil - \sum_{i\in I\setminus J} \frac{c_i}{c} \right) &= \left\lfloor \frac{c_j}{c} - \left(\frac{A}{c} - \sum_{i\in I\setminus J} \frac{c_i}{c} \right) \right\rfloor \\ &= \left\lfloor \frac{1}{c} \cdot \left(c_j - \left(A - \sum_{i\in I\setminus J} c_i \right) \right) \right\rfloor \geq 0\end{aligned}$$

Similarly, every assignment that produces a clause from ι' also produces a clause from ι.

The same holds (by an easier yet very similar argument) for the multiplication rule. □

Lemma 5. *Let integer inequality ι be an integer linear combination of integer inequalities ι_1 and ι_2, let* $\mathtt{DGF}(\iota_1)$, $\mathtt{DGF}(\iota_2) \leq A$*. Then every clause C of the Boolean representation $\mathcal{B}(\iota)$ (given by Lemma 2) can be derived from $\mathcal{B}(\iota_1) \cup \mathcal{B}(\iota_2)$ in at most 2^{6A-2} resolution steps.*

Proof. We may rewrite our inequalities as follows (here x_i, y_i, z_i denote literals):

$$\begin{aligned}\iota_1 : \quad &\sum_1^N e_i' z_i + \sum_1^K a_i x_i + \sum_1^L d_i y_i \geq A_1, \\ \iota_2 : \quad &\sum_1^N e_i'' z_i + \sum_1^K b_i (1 - x_i) + \sum_1^L d_i (1 - y_i) \geq A_2, \\ \iota : \quad &\sum_1^N e_i z_i + \sum_1^K (a_i - b_i) x_i \geq A_1 + A_2 - \sum_1^K b_i - \sum_1^L d_i.\end{aligned}$$

Here all coefficients are strictly positive, possibly except for some of the e'_i's and e''_j's, which are nonnegative. In other words, Z contains literals that are not cancelled by the application of the addition rule, X contains literals that are partially cancelled, and Y contains literals that are cancelled completely. We denote $X = \{x_1, \ldots, x_K\}$, $Y = \{y_1, \ldots, y_L\}$, $Z = \{z_1, \ldots, z_N\}$. Also denote $\overline{S} = \{\overline{s} \mid s \in S\}$ for any set S.

By Lemma 3 there exists a resolution proof Π of C from the clauses of $\mathcal{B}(\iota_1) \cup \mathcal{B}(\iota_2)$. Note that $\overline{Z} \cap C = \emptyset$ and $\overline{Z} \cap D = \emptyset$ for every $D \in \mathcal{B}(\iota_1) \cup \mathcal{B}(\iota_2)$. Hence, Lemma 3 provides Π that does not contain any negative occurrences of z_i's. Let π be the assignment that turns ι into C; denote $Z_\pi = \{z \in Z \mid \pi(z) = 1\}$ and $Z' = Z \setminus Z_\pi$. Note that $Z' \subseteq C$. Therefore, if one adds Z' to each clause in Π, the proof will remain a valid proof of C from the clauses $D^*_i = D_i \cup Z'$, where $D_i \in \mathcal{B}(\iota_1) \cup \mathcal{B}(\iota_2)$. Note that $|X| + |Y| + |Z_\pi| < 2A$; otherwise $\mathtt{DGF}(\iota|_\pi)$ would be non-positive, and the clause C would be a constant `True`. There are at most $2^{3|X \cup Y \cup Z_\pi|} \leq 2^{6A-3}$ possible clauses of the form $Z' \cup T$, where $T \subseteq X \cup \overline{X} \cup Y \cup \overline{Y} \cup Z_\pi$, hence the modified (dag-like) version of the proof Π cannot contain more than 2^{6A-3} clauses. It remains to add at most 2^{6A-3} steps needed to obtain D^*_i's from D_i's by the weakening rule. □

Theorem 1. *A Cutting Plane proof Π with $\max_{\iota \in \Pi} \mathtt{DGF}(\iota) \leq d \leq n/2$ of a formula in CNF with n variables can be transformed into a resolution proof of size at most $\binom{n}{d-1}|\Pi|2^{6d}$.*

Proof. Each step $\dfrac{\iota_1,\ \iota_2}{\iota}$ or $\dfrac{\iota_1}{\iota}$ of Π can be replaced by at most $\binom{n}{d-1}2^{6d-2}$ resolution steps inferring the $\binom{n}{d-1}$ (see Lemma 2) possible clauses of $\mathcal{B}(\iota)$ from $\bigcup_i \mathcal{B}(\iota_i)$, by a 2^{6d-2}-length resolution proof each. (For addition steps such a resolution proof is given by Lemma 5, for other steps it is not needed by Lemma 4.) □

Corollary 1. *If formulas F_n (where F_n contains n variables) have no resolution proofs containing less than $2^{c_{\mathrm{res}}n}$ clauses ($c_{\mathrm{res}} > 0$ is a constant), then these formulas do not have Cutting Plane proofs of size less than $2^{c_{\mathrm{CP}}n}$ and degree of falsity bounded by $c_{\mathtt{DGF}}n$ for every choice of positive constants $c_{\mathrm{CP}} < c_{\mathrm{res}}$ and $c_{\mathtt{DGF}} \leq \frac{1}{2}$ such that*

$$c_{\mathrm{CP}} + 6c_{\mathtt{DGF}} - c_{\mathtt{DGF}} \log_2 c_{\mathtt{DGF}} - (1 - c_{\mathtt{DGF}}) \log_2(1 - c_{\mathtt{DGF}}) \leq c_{\mathrm{res}}. \tag{5}$$

In particular, formulas F_n have only exponential-size Cutting Plane proofs of degree of falsity bounded by an appropriate linear function of n.

Proof. By Theorem 1 Cutting Plane proofs of size less than $2^{c_{\mathtt{DGF}}n}$ could be converted into resolution proofs of size less than

$$\binom{n}{c_{\mathtt{DGF}}n-1}2^{c_{\mathrm{CP}}n+6c_{\mathtt{DGF}}n} = o\big(2^{(c_{\mathrm{CP}}+6c_{\mathtt{DGF}}-c_{\mathtt{DGF}}\log_2 c_{\mathtt{DGF}}-(1-c_{\mathtt{DGF}})\log_2(1-c_{\mathtt{DGF}}))n}\big) = o\big(2^{c_{\mathrm{res}}n}\big)$$

(the first equality uses Stirling's formula).

Finally, note that $f(c) = 6c - c\log_2 c - (1 - c)\log_2(1 - c)$ decreases to 0 as c decreases from $\frac{1}{2}$ to 0. Therefore, for every $c_{\mathrm{CP}} < c_{\mathrm{res}}$ there is $c_{\mathtt{DGF}}$ that satisfies (5). □

Corollary 2. *There exists a positive constant δ such that Tseitin-Urquhart formulas of n variables (described in [Urq87]) have only $2^{\Omega(n)}$-size Cutting Plane proofs with degree of falsity bounded by δn.*

Proof. Follows immediately from Corollary 1 and Urquhart's theorem:

Theorem. ([Urq87–Theorem 5.7]) *There is a constant $c > 1$ such that for sufficiently large m, any resolution refutation of S_m contains c^n distinct clauses, where S_m is of length $O(n)$, $n = m^2$.* □

5 Further Research

A straightforward open question is to prove an exponential lower bound on the lengths of **CP** refutations of Tseitin tautologies without the restriction on the degree of falsity. One way to do it could be to characterize the inequalities that follow from subformulas of these tautologies and have a large degree of falsity.

References

[BGIP01] Sam Buss, Dima Grigoriev, Russell Impagliazzo, and Toniann Pitassi. Linear gaps between degrees for the polynomial calculus modulo distinct primes. *JCSS*, 62:267–289, 2001.

[BSW01] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow – resolution made simple. *JACM*, 48(2), 2001.

[CR79] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(1):36–50, March 1979.

[Das02] Sanjeeb Dash. Exponential lower bounds on the lengths of some classes of branch-and-cut proofs. IBM Research Report RC22575, September 2002.

[GHP02] Dima Grigoriev, Edward A. Hirsch, and Dmitrii V. Pasechnik. Complexity of semialgebraic proofs. *Moscow Mathematical Journal*, 2(4):647–679, 2002.

[Goe91] Andreas Goerdt. The Cutting Plane proof system with bounded degree of falsity. In *Proceedings of CSL 1991*, volume 626 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 1991.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[LS91] L. Lovász and A. Schrijver. Cones of matrices and set-functions and 0-1 optimization. *SIAM J. Optimization*, 1(2):166–190, 1991.

[Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.

[Raz85] A. A. Razborov. Lower bounds on the monotone complexity of some boolean functions. *Doklady Akad. Nauk SSSR*, 282:1033–1037, 1985.

[Rob68] J. A. Robinson. The generalized resolution principle. *Machine Intelligence*, 3:77–94, 1968.

[Tse68] G. S. Tseitin. On the complexity of derivation in the propositional calculus. *Zapiski nauchnykh seminarov LOMI*, 8:234–259, 1968. English translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.

[Urq87] Alasdair Urquhart. Hard examples for resolution. *JACM*, 34(1):209–219, 1987.

Resolution Tunnels for Improved SAT Solver Performance

Michal Kouril and John Franco

University of Cincinnati, Cincinnati, OH 45221-0030, USA
mkouril@ececs.uc.edu
http://www.ececs.uc.edu/~mkouril

Abstract. We show how to aggressively add uninferred constraints, in a controlled manner, to formulae for finding Van der Waerden numbers during search. We show that doing so can improve the performance of standard SAT solvers on these formulae by orders of magnitude. We obtain a new and much greater lower bound for one of the Van der Waerden numbers, specifically a bound of 1132 for $W(2, 6)$. We believe this bound to actually be the number we seek. The structure of propositional formulae for solving Van der Waerden numbers is similar to that of formulae arising from Bounded Model Checking. Therefore, we view this as a preliminary investigation into solving hard formulae in the area of Formal Verification.

1 Introduction

Resolution is a general procedure that may be used to determine whether a given CNF expression has a model and to supply a certificate of unsatisfiability if it doesn't. The idea predates the often cited work reported in [22] and for decades resolution has been one of the primary engines for CNF SAT solvers. In the last 10 years tree resolution, in the form of variants of DPLL [10], has given way to DAG resolution through the introduction of clause learning and recording during search. This and other ideas have led to a spectacular improvement in the performance of resolution-based SAT solvers. Hardware and algorithmic improvements have together contributed to perhaps an order 10^9 speed-up in SAT solving over the past 15 years and, consequently, some problems considered very difficult then are now considered trivial. But there remain many problems which are considered hard, for example in the important domains of protocol and hardware verification.

The last 15 years has also seen some brilliant theoretical work that has revealed exponential lower bounds for tree and DAG resolution, and has illuminated the reasons for it, when resolution is applied to "sparse," unsatisfiable CNF formulae (e.g. [2, 3, 4, 8, 13, 18, 29]). In such cases, very large resolvents must be created first, then resolved to get the smaller clause constraints that play a significant role in establishing the refutation. Generating the large resolvents is expensive, particularly since exponentially many have to be generated.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 143–157, 2005.

In DPLL terms, this means a search space of great breadth may have to be explored. Metaphorically, we may think of search breadth as a mountain that must be climbed; on the other side of the mountain the search breadth may be significantly reduced due to the short resolvents that are finally generated. Clearly, we need to find some way to "tunnel" through this mountain and arrive quickly in the fertile valley of low-breadth search space, if it exists. It is the aim of this paper to explore this possibility for Boolean expressions with a particular structure.

In the Satisfiability literature, the term "tunnel" has been applied to Stochastic Local Search algorithms, a class which includes members of the WalkSAT, GSAT, and other families [14,16,17,20,25]. In that context, one may think of a location as an assignment of values to variables and the height at a location as the number of constraints falsified by the assignment at that location. Then, for a given, hard Boolean expression, there are generally many mountain peaks and the objective is to locate and enter the deepest valley. Changing the value of a single variable merely moves the current location up and down the side of a mountain but changing values of several variables permits "tunneling" into another valley. This use of tunneling is to be distinguished from the way we use it: in our context there is one principal mountain to get over or around and the valley on the other side can be quite wide. To reach the top of our mountain one must wait for many large constraints to be learned. But, in many cases, we cannot afford to wait: to reach the valley in a reasonable time, *constraints must be efficiently added before they are learned.*

There are several ways to do this, some safe and some risky. A reasonably efficient method for finding a safe tunnel, which is actually more like a cut, through the mountain has been identified in [31] for a class of non-CNF formulae. Given Boolean formulae $b_1, b_2, ..., b_m$, let $\phi = b_1 \wedge \cdots \wedge b_m$ and let $V' = \{v_1, \cdots, v_k\}$ be a subset of variables occurring in ϕ. Re-index the formulae so that at least one variable of V' occurs in b_i for all $1 \leq i \leq n$, and no variables of V' occur in b_i, for $n < i \leq m$. Let $\mathcal{M}' = \{M'_1, \cdots, M'_{2^k}\}$ be the set of all possible truth assignments to the variables in V'. Write $b_j|_{M'_i}$ or $\phi|_{M'_i}$ to mean the variables of b_j or ϕ, respectively, are assigned values according to assignment M'_i, or are left unassigned if they are not assigned in M'_i.

Theorem 1. ([31]) *For every* $1 \leq i \leq 2^k$*, define*

$$\delta_i = \bigvee_{1 \leq j \leq n} (\overline{b_j|_{M'_i}} \wedge \bigvee_{\substack{1 \leq s \leq 2^k \\ s \neq i}} b_j|_{M'_s}).$$

Then, for any $1 \leq i \leq 2^k$*, if* $\delta_i \equiv$ `False` *then* $\phi|_{M'_i}$ *is satisfiable if and only if* ϕ *is satisfiable.* ∎

According to Theorem 1, it may be possible to assign values to a particular set of variables in such a way that the satisfiability of ϕ is unaffected. Those values may not be inferred at all, but they are nevertheless *safe* to assign and doing so reduces ϕ somewhat. A safe assignment is a generalization of the notion of *autarky* [21], defined for CNF formulae, to formulae that are conjunctions of arbitrary Boolean formulae. The test implied in the theorem can be conducted with reasonable efficiency (see [31] for details).

Although safe assignments can be useful, substantial decreases in computational effort require more aggressive use of uninferred constraints. This need not lead to errors: a constraint can be retracted during search if it is inferred `False`. The field of non-monotonic reasoning has provided many insights on how this might be done. Of course, we would like to be able to add constraints aggressively without having to worry about retracting them and, as we show in this paper, this can sometimes be done to achieve a result which is an approximation to the actual result. In other words, the extra constraints may prevent an optimal solution from being returned but are weak enough to admit suboptimal solutions that are not far from optimal. But, if the extra constraints are too weak, a solver may take too much time and not find a good suboptimal result. So, we need a good "heuristic" for adding constraints in some optimal way.

At this point in time, it seems the spectacular tunneling success we seek, which will be a consequence of our choice of tunnel heuristic, is practical only by a careful analysis of the specific structure of a given formula. In this paper we underscore this point by developing aggressive tunnel heuristics for formulae associated with the problem of finding Van der Waerden numbers (described in the next section). Such formulae are currently extremely difficult for off-the-shelf SAT solvers, even though most of them are satisfiable. By adding aggressive tunnel constraints to the formulae, however, we are able to find the best bound yet, by far, for $W(2,6)$. Our analysis serves as an example for attempting aggressive tunnel heuristics for other hard problems.

2 Van der Waerden Numbers and Satisfiability

Van der Waerden numbers arise from a set partition problem [30]. Partition the set $S_n = \{1, ...n\}$ of the first n positive consecutive integers into k classes. Let $P_{n,k}(l)$ be a proposition that is `True` if and only if all partitions of S_n into k classes contain at least one arithmetic progression of length l in at least one class. The k,l Van der Waerden number, denoted $W(k,l)$, is the minimum n for which $P_{n,k}(l)$ is `True`.

There is no known closed form expression for $W(k,l)$ and all but five of the first few numbers are unknown. Table 1 shows all the known Van der Waerden numbers. In 1979 $W(4,3)$ became the most recent addition to this table.

Table 1. Known Van der Waerden numbers

$k \setminus l$	**3**	**4**	**5**
2	9	35	178
3	27		
4	76		

Upper and lower bounds on some of the remaining numbers have been derived but they are so far apart that they are of little practical use. An unpublished general upper bound is [32]

$$W(k,l) \leq e^{e^{(1/k)e^{e^{(l+110)}}}},$$

and a general lower bound, due to the Lovász local lemma, is [32]

$$W(k,l) > \left(\frac{k^l}{elk}\right)(1+o(1)).$$

Work on specific Van der Waerden numbers has sharpened some of these bounds as the results of Table 2 (taken from [11]) show.

The number $W(k,l)$ can be found by determining whether solutions exist for certain formulae of a class of CNF formulae described in Table 3. We refer to a formula of this class, with parameters n, k, l, by $\psi^n_{k,l}$. A solution exists for $\psi^n_{k,l}$ if and only if $n < W(k,l)$. So, several of these formulae may be solved for various values of n until that boundary is reached. In this manuscript $\psi^n_{k,l}$ is treated as a set of clauses to make some algorithmic operations easier to express.

The nature of $\psi^n_{k,l}$ is such that the number of solutions increases with n up to a point, then decreases. The formulae, as expected, become more difficult in

Table 2. Known bounds on van der Waerden numbers

$k \setminus l$	**3**	**4**	**5**	**6**	**7**	**8**
2	9	35	178	> 695	> 3702	> 7483
3	27	> 291	> 1209	> 8885	> 43854	> 161371
4	76	> 1047	> 10436	> 90306	> 262326	
5	> 125	> 2253	> 24044	> 177955		
6	> 206	> 3693	> 56692			

Table 3. Formula $\psi^n_{k,l}$ for finding Van der Waerden numbers. Equivalence classes are named $C_1, C_2, ..., C_k$ for convenience

Variables	**Subscript Range**	**Meaning**
$v_{i,j}$	$1 \leq i \leq n, 1 \leq j \leq k$	$v_{i,j} \equiv 1$ iff $i \in C_j$

Clauses	**Subscript Range**	**Meaning**
$\{\bar{v}_{i,r}, \bar{v}_{i,s}\}$	$1 \leq i \leq n, 1 \leq r < s \leq k$	i is in at most one class
$\{v_{i,1}, \ldots, v_{i,k}\}$	$1 \leq i \leq n$	i is in at least one class
$\{\bar{v}_{r,j}, \bar{v}_{r+1,j}, \ldots, \bar{v}_{r+l-1,j}\}$	$1 \leq r \leq n-l+1$	
$\{\bar{v}_{r,j}, \bar{v}_{r+2,j} \ldots, \bar{v}_{r+2(l-1),j}\}$	$1 \leq j \leq k$	no arithmetic progression
...	...	of length l in C_j
$\{\bar{v}_{r,j}, \bar{v}_{r+t,j} \ldots, \bar{v}_{r+t(l-1),j}\}$	$t = \lfloor (n-r)/(l-1) \rfloor$	

Table 4. Bounds on van der Waerden numbers obtained by SAT solvers ([11])

$k \setminus l$	**3**	**4**	**5**	**6**	**7**	**8**
2	9	35	178	> 341	> 614	> 1322
3	27	> 193	> 676	> 2236		
4	76	> 416				
5	> 125	> 880				
6	> 194					

the latter range. This difficulty may prevent the boundary from being reached or even approached. In that case, there is no choice but to accept a lower bound which is the largest n for which a solution to $\psi^n_{k,l}$ is found.

SAT solvers have been applied to the above formulations with the results shown in Table 4 (taken from [11]), all lower bounds. Except in one case, all these bounds are greatly inferior to those obtained analytically.

We are interested in $W(2,6)$. When we employ aggressive tunneling techniques, we can push the lower bound to 1132 from the previously known best value of 696 [28] [27] shown in Table 2 which itself far exceeds the best prior bound of 342 obtained by SAT solvers and shown in Table 4. The reason we have a bound instead of the actual number is that aggressive tunneling forces our SAT solver to be incomplete. We emphasize that although an incomplete solver precludes finding a refutation for the formula, it does provide an apparently tight bound for $W(2,6)$. We believe the bound is tight because it is obtained using three widely different tunnels. In addition, using tunnels aggressively we have found a bound for $W(3,4)$ which matches the best analytic bound shown in Table 2 and greatly exceeds the best bound obtained by SAT solvers as shown in Table 4. This is further evidence of the tightness of bound in the presence of these aggressive tunnels.

Our interest in examining tunnels for Van der Waerden numbers is due partly to this being an interesting problem to many mathematicians but mainly because the propositional formulae for solving Van der Waerden numbers have a structure that is similar to formulae found in Bounded Model Checking and other practical applications. Therefore, we view this work as a preliminary to investigations on problems in the area of Formal Verification, among others.

Formulae showing structural similarities to Van der Waerden formulae do so, in part, because their corresponding problems are typically circuit queries with a time dependent nature, such as verification problems. In such problems a circuit is fixed so there are numerous repetitions of subformulae, each corresponding to the circuit properties at a different time step. The subformulae differ in the variables they contain. This structure is characteristic of the Van der Waerden formulae shown in Table 5. This structure sometimes is partly responsible for difficult formulae since the making of small inferences is delayed considerably in such cases.

3 Formulation for $W(2,6)$ and the Tunnels

3.1 Formulation

Since we consider a case where $k = 2$, we reinterpret variables to remove some of the constraints shown in Table 3. The formulae we consider are described in Table 5. They use single index variables. For puposes of discussion, variable indices have been translated so that v_0 and v_1 are the middle variables. In doing so, the number of variables is always even. It is straightforward to consider odd variable formulae as well and we leave this for the reader. In what follows, n is even when we consider formulae $\psi^n_{2,6}$.

Table 5. Formula $\psi^n_{2,6}$, n even, for finding $W(2,6)$. Classes are named C_1, C_2 for convenience

Variables	Subscript Range	Meaning
v_i	$-n/2 < i \leq n/2$	$v_i \equiv 1$ if $i + n/2 \in C_1$ $v_i \equiv 0$ if $i + n/2 \in C_2$

Clauses	Subscript Range	Meaning
$\{\bar{v}_i, \bar{v}_{i+1}, \dots, \bar{v}_{i+5}\}$	$-n/2 < i \leq n/2 - 5$	no arithmetic progression of length 6 in C_1
$\{\bar{v}_i, \bar{v}_{i+2} \dots, \bar{v}_{i+10}\}$		
...	...	
$\{\bar{v}_i, \bar{v}_{i+t} \dots, \bar{v}_{i+5t}\}$	$t = \lfloor (n/2 - i)/5 \rfloor$	
$\{v_i, v_{i+1} \dots, v_{i+5}\}$	$-n/2 < i \leq n/2 - 5$	no arithmetic progression of length 6 in C_2
$\{v_i, v_{i+2} \dots, v_{i+10}\}$		
...	...	
$\{v_i, v_{i+t} \dots, v_{i+5t}\}$	$t = \lfloor (n/2 - i)/5 \rfloor$	

3.2 Motivating the Use of a Tunnel - Analyzing $\psi^n_{2,l}$

The approach to finding $W(2,6)$ that is described below is the result of an analysis of the performance of an off-the-shelf SAT solver on formulae $\psi^n_{2,l}$ (Table 3), and patterns of variable assignments satisfying those formulae.

Figure 1 shows SAT solver performance on $\psi^n_{2,5}$, for various values of n. The vertical axis measures the number of nodes of the search space at the depth indicated by the horizontal axis. In all cases, the entire search space was explored, even if the input formula was satisfiable, but the results are similar if the solver stops immediately upon discovering a solution. Although the displayed results have been obtained using stock settings, similar results, which are not shown, apply for various settings of SAT solver parameters. According to Figure 1 there is a performance mountain that has roughly the same shape, regardless of n. The mountain tails off to a point of little significance after rising to a formidible peak. In addition to the mountain is a smaller peak which behaves more like a wave since it always appears near n.

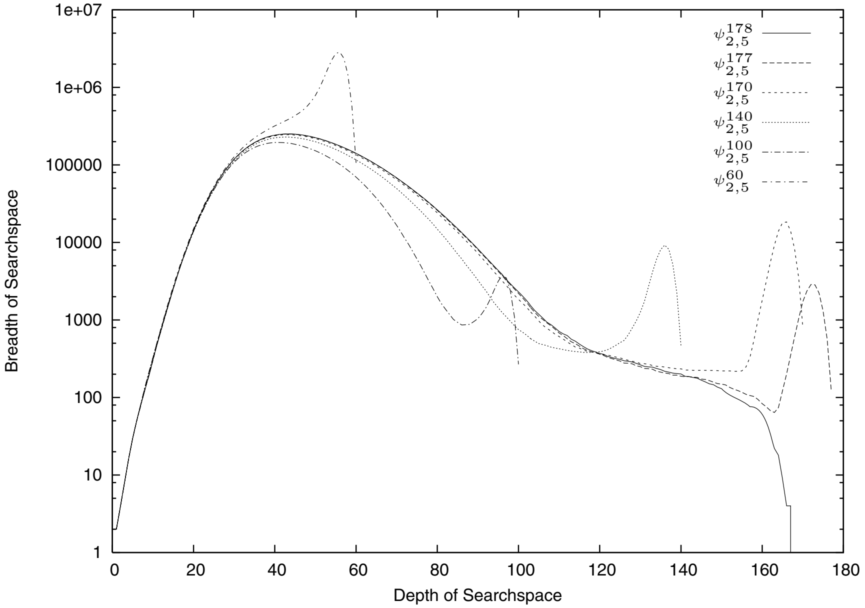


Fig. 1. Typical SAT solver performance on $\psi_{2,5}^{n}$ for various values of n. Each curve shows the breadth of the search space at the search depth indicated on the horizontal. Curves shown are due to SBSAT which is a SAT solver currently under development at the University of Cincinnati. Other SAT solvers have been shown to perform similarly

From performance curves and known Van der Waerden numbers and bounds we have observed the following:

Observation 1. Consider a performance plot of search breadth vs. depth for any common SAT solver applied to $\psi_{2,l}^{n}$. Such a plot has two maxima, the greatest of which occurs at approximately the same depth, say $W(2,l)/(2(l-1))$, for $n > W(2,l)/(l-1)$. The value of the greatest maximum is approximately the same for $n > W(2,l)/(l-1)$ and several orders of magnitude greater than the search breadth at depth $W(2,l)/(l-1)$.

Observation 2. $W(2,l) \approx l * W(2,l-1)$, at least for small l.

From the above observations we propose a way to solve the very difficult $W(2,6)$ formulae. Start by searching for solutions to $\psi_{2,6}^{n_o}$ where n_o is large enough so that the mountain *can* be crossed but smaller than the suspected value of $W(2,6)$. By Observations 1 and 2, n_o should be greater than about 210. To get through the mountain, append a *tunnel* to the formula. We have developed three applicable tunnels which are specially designed to take advantage of certain structural or analytical characteristics of the formulae and are described separately in Section 3.4. The mountain is crossed when the search breadth is considered "low." From then on, by Observation 1, search is accomplished efficiently no matter what. Retract the tunnel. Add clauses so that the solver is effectively seeing $\psi_{2,6}^{n_o+x}$ for $x = 1, 2, 3, ...$ until search breadth falls to 0. The depth at which this occurs is a bound on $W(2, 6)$.

Details are given in the following sections.

3.3 Procedure for Finding a Bound on $W(2, 6)$

We used a SAT solver designed specifically to solve formulae $\psi_{2,l}^{n}$, and which incorporates special optimizations as outlined in Section 3.6. The outline below briefly describes the search process using the special solver.

1. The solver is started on the input $\psi_{2,6}^{n_o}$, for some even n_o, plus a collection of clauses representing the tunnel. The parameter n_o is determined according to the description given in Section 3.2. There are three types of tunnels that have been applied. Detailed descriptions are given in Section 3.4. All three yield the same lower bound on $W(2, 6)$ as stated in Section 3.4. Other tunnels are possible but were not tried.
2. Instead of a depth-first, or even priority-driven evaluation of the search space, as is commonly practiced, the solver conducts a strictly breadth-first evaluation. We chose breadth-first search to find all solutions (not just the first one) so we can at the point of n_o remove the tunnel and continue the search by extending the existing set of solutions. The order in which variables are considered for evaluation is fixed for all branches of the search tree, regardless of values assigned previously. The order represents a reflection around the center variable and is given as follows, from left to right: $v_0, v_1, v_{-1}, v_2, v_{-2}, ...$. This is undoubetdly an inferior choice with respect to building the entire search space. Reasons for this choice are given in Section 3.5.
3. The search reaches depth n_o because there are so many solutions to $\psi_{2,6}^{n_o}$ and the tunnel constraints do not filter some of them. At this depth all variables of $\psi_{2,6}^{n_o}$ have been assigned values on all leaves of the search tree. Clauses from the set $\psi_{2,6}^{n_o+1} \setminus \psi_{2,6}^{n_o}$ are then added to the clause database of the solver and the search commences as before. The tunnel is retracted.

   Remark: The tunnel has been designed so that the mountain has been crossed, for the most part, by this time. From this point on, the breadth of the search space is moderately small because the values of many variables are inferred on all branches, so the search continues quickly.

4. The following is repeated for $m = 2, 3, ...$ until the search breadth becomes 0: when the search depth reaches $n_o + m - 1$, clauses from $\psi_{2,6}^{n_o+m} \setminus \psi_{2,6}^{n_o+m-1}$ are added to the solver's clause database and the search continues. When the search breadth becomes 0, a lower bound of $n_o + m$ is found for $W_{2,6}$.

   Remark: Upon completion of every iteration of this step, a non-zero search breadth means at least one solution for $\psi_{2,6}^{n_o+m-1}$ exists, hence $n_o + m$ is a lower bound for $W(2, 6)$.

   Remark: There is no clause recording. Experiments show that zChaff, for example, does not benefit from clause recording on this family of formulae.

> We believe this is because the structure of the formulae is such that inferences can only be determined at high search depth. This is why we needed the tunnels in the first place.

With the above modifications to the SAT solver, and the improved formulation shown in Table 5, a greatly improved bound for $W(2,6)$ was obtained. However, this was not the case if no tunnels had been added at the outset.

Below we present performance figures for zChaff (v.2004.11.15) and Berkmin (v.561) as well as the special SAT solver described above. It is important to realize that zChaff and Berkmin can succeed with tunnels only if some mechanism is provided for determining when the other side of the mountain is reached. The mechanism we used was to apply our special solver up to Step 3. Then we removed the tunnel, and sequentially applied zChaff or Berkmin to the given formula for *every* partial assignment not falsified at depth n_o until either a solution was found or it was determined that no solution was possible.

3.4 The Tunnels

Three different aggressive tunneling techniques to improve SAT solver performance are described in the subsections below. All three yield the same new lower bound of 1132 for $W(2,6)$. This is significant for two reasons: 1) this is a big improvement over the previous best bound of 696 [28] [27]; 2) since three different techniques stopped at the same point, we conjecture $W(2,6) = 1132$.

First Tunnel. The first tunnel arises from an analysis of solutions to $\psi_{2,l}^{W(2,l)-1}$ (recall $W(2,l)-1$ is the greatest n for which a solution to $\psi_{2,l}^{n}$ exists). Figures 2 and 3 help visualize patterns associated with a typical solution to $\psi_{2,4}^{34}$ and $\psi_{2,5}^{177}$, respectively. For both figures, the solution is shown as a sequence of 0's and 1's representing an assignment of values to variables in increasing order of index, from left to right. Each curve shown is called a *solution curve* and is derived from the solution in the figure. A solution curve rises one unit for every 1 encountered and drops one unit for every 0 encountered when traversing the solution from left to right. Observe that the number of peaks in each figure is $l-1$ and, more importantly, there appears to be a limited length pattern of reverse symmetry, which we call a *reflected pattern*, in the vicinity of at least one of the peaks.

We conjecture the following based on observations such as those depicted in Figures 2 and 3:

Conjecture 1. For every $\psi_{2,l}^{W(2,l)-1}$ there exists a solution that contains at least one reflected pattern of length $W(2,l)/((l-1)*2)$ with the middle positioned somewhere between $W(2,l)/(l-1)$ and $W(2,l)*(l-2)/(l-1)$.

The tunnel is designed as a filter for consecutive variable assignment patterns that are not reverse symmetric. The tunnel consists of clauses involving s consecutive variables where, s is even and by Conjecture 1 and Observation 2, $s < \lfloor 1068/10 \rfloor = 106$. We tried several values for s including 60, 80, 100, even 150 and all worked, but speed increased significantly with increasing s up to 150.

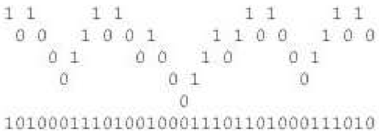


Fig. 2. Typical solution curve for $\psi_{2,4}^{34}$. This is the largest satisfiable formula for $W(2,4)$

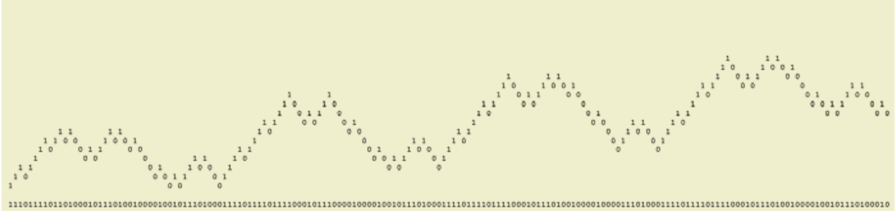


Fig. 3. Typical solution curve for $\psi_{2,5}^{177}$. This is the largest satisfiable formula for $W(2,5)$. For both figures, the solution is shown as a sequence of 0's and 1's representing an assignment of values to variables in increasing order of index, from left to right. The top of each curve rises one unit for every 1 encountered and drops one unit for every 0 encountered when traversing the solution from left to right

Table 6. First tunnel constraints added to $\psi_{2,6}^{n_o}$ initially, then retracted after "tunneling"

Tunnel Clauses	Subscript Range	Meaning
$\{v_{-i}, v_{i+1}\}, \{\bar{v}_{-i}, \bar{v}_{i+1}\}$	$0 \leq i < s/2$	force $v_{-i} \equiv \bar{v}_{i+1}$.

The first family of tunneling constraints is shown in Table 6. By using negative indices in Table 5 these constraints remain fixed as n grows. Otherwise, the tunnel would have to move with n.

Second Tunnel. From the results obtained by using the first tunnel alone, it was observed that some small assignment patterns *did not* occur in solutions. The second tunnel filters those patterns. This action is opposite to that of *forcing* patterns to occur which is the objective of the first tunnel. Consequently, the first tunnel spans a small number of variables because longer forced reverse symmetric patterns do not exist in any solution to $\psi_{2,6}^{W(2,6)-1}$, but the second tunnel spans all variables because non-solution patterns can appear anywhere in the clauses of $\psi_{2,6}^{W(2,6)-1}$.

The second family of tunneling constraints is shown in Table 7. The maximum value of 20 for t is a compromise: the tunnel needs to be big enough to have an impact but small enough to keep some solutions around to the end. The number 20 was determined by experiment on $\psi_{2,6}^{n}$ formulae.

Table 7. Second tunnel constraints added to $\psi_{2,6}^{n_o}$ initially, then retracted after "tunneling"

Tunnel Clauses	**Subscript Range**	**Filters**
$\{v_i, \bar{v}_{i+t}, v_{i+2t}, \bar{v}_{i+3t}, v_{i+4t}, \bar{v}_{i+5t}\}$	$-n/2 < i \leq n/2 - 5t$	010101
$\{\bar{v}_i, v_{i+t}, \bar{v}_{i+2t}, v_{i+3t}, \bar{v}_{i+4t}, v_{i+5t}\}$	$1 \leq t \leq 20$	101010
$\{v_i, v_{i+t}, \bar{v}_{i+2t}, \bar{v}_{i+3t}, v_{i+4t}, \bar{v}_{i+5t}, \bar{v}_{i+6t}, v_{i+7t}\}$		00110110
$\{\bar{v}_i, \bar{v}_{i+t}, v_{i+2t}, v_{i+3t}, \bar{v}_{i+4t}, v_{i+5t}, v_{i+6t}, \bar{v}_{i+7t}\}$		11001001
$\{v_i, \bar{v}_{i+t}, \bar{v}_{i+2t}, v_{i+3t}, \bar{v}_{i+4t}, \bar{v}_{i+5t}, v_{i+6t}, v_{i+7t}\}$		01101100
$\{\bar{v}_i, v_{i+t}, v_{i+2t}, \bar{v}_{i+3t}, v_{i+4t}, v_{i+5t}, \bar{v}_{i+6t}, \bar{v}_{i+7t}\}$	$-n/2 < i \leq n/2 - 7t$	10010011
$\{v_i, v_{i+t}, \bar{v}_{i+2t}, \bar{v}_{i+3t}, \bar{v}_{i+4t}, v_{i+5t}, v_{i+6t}, \bar{v}_{i+7t}\}$	$1 \leq t \leq 20$	00111001
$\{\bar{v}_i, \bar{v}_{i+t}, v_{i+2t}, v_{i+3t}, v_{i+4t}, \bar{v}_{i+5t}, \bar{v}_{i+6t}, v_{i+7t}\}$		11000110
$\{\bar{v}_i, v_{i+t}, v_{i+2t}, \bar{v}_{i+3t}, \bar{v}_{i+4t}, \bar{v}_{i+5t}, v_{i+6t}, v_{i+7t}\}$		10011100
$\{v_i, \bar{v}_{i+t}, \bar{v}_{i+2t}, v_{i+3t}, v_{i+4t}, v_{i+5t}, \bar{v}_{i+6t}, \bar{v}_{i+7t}\}$		01100011

Third Tunnel. In [27] solutions to $\psi_{2,6}^{n}$ are found for various values of n including 565 and 695. The latter number implies the best known analytic lower bound on $W(2,6)$. Regarding the solution to $\psi_{2,6}^{565}$, re-index the assigned variables

$$v_{-282}, ..., v_0, v_1, ...v_{282}$$

to

$$v_{-564}, v_{-562}, ..., v_0, v_2, ..., v_{562}, v_{564}$$

and add unassigned variables

$$v_{-565}, v_{-563}, ..., v_1, ..., v_{563}, v_{565}.$$

Observe that this operation will not introduce any arithmetic progression among the even indexed variables. The assignment to the even indexed variables is the third tunnel. Search for a solution to $\psi_{2,6}^{1131}$ among the variables above, keeping the values of the even indexed variables fixed. Search completes in a reasonable time with a solution.

Remarkably, using any of the three tunnels or even combining them leads to the same bound of 1132 for $W(2,6)$.

3.5 Choosing the Search Heuristic

Variables are always considered in the following order, regardless of assignment:

$$v_0, v_1, v_{-1}, v_2, v_{-2}, v_3, v_{-3}, \cdots$$

The reason is that by assigning values to middle variables first, there is a good chance of inferences developing, symmetrically, for higher and lower indexed variables. If, say, the lower indexed variables were assigned first, perhaps half of the potential future inferences would not be realized early. This is particularly important for the first tunnel where the inferences are needed to appear outside of the tunnel variables.

3.6 Optimizations and Special Procedures

We were able to get the lower bound of 1132 for $W(2,6)$ with either tunnel using the special solver described in Section 3.3 with the optimizations mentioned below. However, for stock SAT solvers we could not get this result in reasonable time using either the first or second tunnels alone but could get the result using both simultaneously.

The following optimizations to the special solver were used.

1. Data structures specifically designed for very fast checking of arithmetic progressions and inferences were used. Estimated speed up due to these structures is about a factor of 25. Another factor of approximately 2 speedup is due to item 3 below.
2. Without optimization, all nodes of the search space would have two children representing `True` and `False` assignments to the variable of that node. However, we generate two children only when inferences force that to be necessary. In other words, we implement a primitive form of conflict-analysis but do not record the result as a clause as is done in modern SAT solvers. Estimated speed up due to this optimization is about a factor of 2.
3. The static search heuristic described in Section 3.5 accounts for an estimated factor of 2 speed up.

4 Performance Results

On $W(2,5)$ formulae, our solver with special data structures took 1 second with the first tunnel of 6 variable width on a 2 GHz Pentium 4 computer and made 143184 variable assignments. Without a tunnel our solver with special data structures took 8 seconds on the same machine and made 719640 variable assignments. On the same machine zChaff (2004.11.15) took 702 seconds and made 570981 assignments and Berkmin (561) took greater than 1500 seconds without a tunnel (see the end of Section 3.3 for information about how zChaff and Berkmin were run). Upon adding the first tunnel of 6 variable width to the Van der Waerden formulae, zChaff (2004.11.15) took 25 seconds and made 93031 assignments. This factor of 30 improvement demonstrates the strength of the idea on the same SAT solver. On $W(2,6)$ formulas zChaff ran out of memory (2GB) in a few hours and BerkMin took greater than 24 hours and was not able to finish, hence this comparison could not be made, only estimated, from the $W(2,5)$ results.

5 Conclusions

We have shown how an analysis of performance curves and solution patterns of a class of CNF formula can present insight for designing effective tunnels through search depth of high breadth. We have chosen to experiment with formulae for finding Van der Waerden numbers since they are a difficult class for standard SAT solvers, apparently because clause recording (learning) is ineffective. By tunneling, reasonable yet uninferred constraints are added just long enough to get to a search depth that has relatively low breadth. Then the tunnel is retracted with the hope that not all solutions have been destroyed by the tunnel. Doing so, we found a solution to $\psi_{2,6}^{1131}$ and therefore a new, significantly improved lower bound on the number $W(2,6)$ of 1132.

The symmetric nature of the formulae and solution played an important part in designing effective tunnels and three tunnels were tried with the same result. This type of symmetry is common in many formula classes that arise from practical applications including problems of formal verification. We believe other difficult problems will succumb to tunneling.

We believe performance curves of search breadth vs. depth, such as shown in Figure 1, provide a "fingerprint" for tunnel effectiveness of problem classes. We speculate the fingerprint will not change much qualitatively from one solver to the next but cannot rule out that possibility. Assuming this, we make some claims about the performance curve with respect to hardness as follows. If a performance curve should reach a peak at very high depth, tunneling is unlikely to be effective. The family of `queue` formulae from bounded model checking seem to fall into this category. We speculate that early, large peaks mean delayed inferences and force exhaustive exploration at search depths corresponding to many unassigned variables.

Although the development of general purpose propositional solvers that work well on all inputs is strongly desireable, at this point it is hard to imagine how this is going to be accomplished. However, a general purpose solver can be assisted greatly by giving special consideration to an input problem class, mining its structure for some property that may be used to improve search. This is what we have done with tunneling. Although we do not forsee generally applicable principals for developing tunnels, we do believe that normal human intuition is enough to uncover exploitable structure in many cases. The Van der Waerden numbers illustrate both points.

Acknowledgements. This work was supported in part by DoD contracts MDA-904-99-C-4547, MDA-904-02-C-1162 and a fellowship grant of the Ohio Board of Regents.

References

1. Akers, S. B.: Binary decision diagrams. IEEE Transactions on Computers **C-27(6)** (1978) 509–516
2. Beame, P., and Pitassi, T.: Simplified and improved resolution lower bounds. Proc. 37th Annual Symposium on Foundations of Computer Science (1996) 274–282.
3. Beame, P., Karp, R. M., Pitassi, T., and Saks, M.: On the complexity of unsatisfiability proofs for random k-CNF formulas. Proc. 30th Annual Symposium on the Theory of Computing (1998) 561–571.
4. Ben-Sasson, E., and Wigderson, A.: Short proofs are narrow - resolution made simple. Journal of the Association for Computing Machinery **48** (2001) 149–169.

5. Brace, K. S., Rudell, R. R., and Bryant, R. E.: Efficient implementation of a BDD package. Proc. 27th ACM/IEEE Design Automation Conf. (1990) 40–45.
6. Bryant, R. E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Comp. **C-35(8)** (1986) 677–691.
7. Burch, J., Clark, E., and Long, D.: Symbolic model checking with partitioned transitions relations. In: Intnl. Conf. on VLSI (Halaas, A., and Denyer, P.B., eds.), IFIP Transactions, North-Holland (1991) 49–58,
8. Chvátal, V., and Szemerédi, E.: Many hard examples for resolution. Journal of the Association for Computing Machinery **35** (1988) 759–768.
9. Cooke, M.: Van der Waerden numbers. Available from `http://home.comcast.net/ rm_cooke/vdw.html`.
10. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. Communications of the Association of Computing Machinery **5** (1962) 394–397.
11. Dransfield, M.R., Liu, L., Marek, V., and Truszczynski, M.: Using Answer-Set Programming to study van der Waerden numbers. Logic and Artificial Intelligence Laboratory, Computer Science Department, College of Enginnering, University of Kentucky (September, 2004). Available from http://cs.engr.uky.edu/ai/vdw/.
12. Dransfield, M., and Bryant, R. E.: Using ordered binary decision diagrams to solve highly structured satisfiability problems. Unpublished technical report CMU-CS-1996, Carnegie Mellon University (1996).
13. Galil, Z.: On resolution with clauses of bounded size. SIAM Journal on Computing 6 (1977) 444–459.
14. Gent, I.P., and Walsh, T.: Towards an understanding of hill-climbing procedures for SAT. Proc. 11th National Conference on Artificial Intelligence (1993) 28–33.
15. Groote, J. F.: Hiding propositional constants in BDDs. Logic Group Preprint Series 120, Utrecht University (1994).
16. Gu, J.: Efficient local search for very large-scale satisfiability problems. ACM SIGART Bulletin **3**(1) (1992) 8–12.
17. Gu, J., Purdom, P.W., Franco, J., and Wah, J.: Algorithms for the Satisfiability problem: a survey. DIMACS Series on Discrete Mathematics and Theoretical Computer Science **35** (1997) 19–151.
18. Haken, A.: The intractability of resolution. Theoretical Computer Science **39** (1985) 297–308.
19. Lee, C. Y.: Representation of switching circuits by binary-decision programs. Bell System Technical Journal **38** (1959) 985–999.
20. McAllester, D., Selman, B., and Kautz, H.A.: Evidence for invariants in local search. Proc. International Joint Conference on Artificial Intelligence (1997) 321–326.
21. Monien, B., Speckenmeyer, E.: Solving satisfiability in less than 2^n steps. Discrete Applied Mathematics **10** (1983) 117–133.
22. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM **12** (1965) 23–41.
23. Pan, G., and Vardi, M. Y.: Search vs. symbolic techniques in satisfiability solving. Proc. Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004).
24. San Miguel Aguirre, A., and Vardi, M. Y.: Random 3-SAT and BDDs: The plot thickens further. In: Principles and Practice of Constraint Programming, (2001) 121-136.
25. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. Proc. 12th National Conference on Artificial Intelligence (1994) 337–343.

26. Somenzi, F.: Colorado University Decision Diagram package. Available from `http://vlsi.colorado.edu/~fabio/CUDD/`.
27. Song, H.Y., Golomb, S.W., and Taylor, H.: Progressions in Every Two-coloration of Z_n. Journal of Combinatorial Theory, Series A **61**(2) (1992) 211–221.
28. Rabung, J.R.: Some Progression-Free Partitions Constructed Using Folkman's Method. Canadian Mathematical Bulletin Vol. **22**(1) (1979) 87–91.
29. Urquhart, A.: Hard examples for resolution. Journal of the Association for Computing Machinery **34** (1987) 209–219.
30. Van der Waerden, B.L.: Beweis einer Baudetschen Veermutung. Nieuw Archief voor Wiskunde **15** (1927) 212–216.
31. Weaver, S.A., Franco, J., and Schlipf, J.S.: Extending Existential Quantification in Conjunctions of BDDs. University of Cincinnati Technical Report.
32. Weisstein, E.W., et al.: van der Waerden Number. ¿From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/vanderWaerdenNumber.html.

Diversification and Determinism in Local Search for Satisfiability*

Chu Min Li[1] and Wen Qi Huang[2]

[1]LaRIA, Université de Picardie Jules Verne,
33 Rue St. Leu, 80039, Amiens Cedex 1, France
chu-min.li@u-picardie.fr
[2]Huazhong university of science and technology, Wuhan, China
wqhuang@mail.hust.edu.cn

Abstract. The choice of the variable to flip in the Walksat family procedures is always random in that it is selected from a randomly chosen unsatisfied clause c. This choice in Novelty or R-Novelty heuristics also contains some determinism in that the variable to flip is always limited to the two best variables in c. In this paper, we first propose a diversification parameter for Novelty (or R-Novelty) heuristic to break the determinism in Novelty and show its performance compared with the random walk parameter in Novelty+. Then we exploit promising decreasing paths in a deterministic fashion in local search using a gradient-based approach. In other words, when promising decreasing paths exist, the variable to flip is no longer selected from a randomly chosen unsatisfied clause but in a deterministic fashion to surely decrease the number of unsatisfied clauses. Experimental results show that the proposed diversification and the determinism allow to significantly improve Novelty (and Walksat).

1 Introduction

Consider a propositional formula $\mathcal{F}$ in Conjunctive Normal Form (CNF) on a set of boolean variables $\{x_1, x_2, \ldots, x_n\}$, the satisfiability problem (SAT) consists in testing whether all clauses in $\mathcal{F}$ can be satisfied by some consistent assignment of truth values to variables.

SAT is the first known [1] and one of the most well-studied NP-complete problems. It has many applications like graph coloring, circuit designing or planning, since such problems can be encoded into CNF formulas in a natural way and solved by a SAT solver.

Given a CNF formula $\mathcal{F}$ and an assignment, local search procedures repeatedly repair locally this assignment, i.e. flipping the value of one variable, to find an assignment satisfying all clauses of $\mathcal{F}$. Since the introduction of GSAT [14] in which the best variable is picked to be flipped to decrease the number of unsatisfied clauses, there has been

* This work is partially supported by Programme de Recherches Avancées de Coooprations Franco-Chinoises (PRA SI02-04) and project Tolérant (Bestfit) under the research program HTSC of the Picardie region in France.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 158–172, 2005.

considerable research on local search methods to find satisfying assignments for CNF formulae, see, e.g. [3, 9, 12, 13, 7, 10, 11, 4].

Perhaps the most significant early improvement was to incorporate a ”random walk” component where variables were flipped from some unsatisfied clause [12], leading to the development of the well-known Walksat procedure [13] in which the variable to flip is always picked from a randomly selected unsatisfied clause. Another contemporary idea was to break ties in favor of least recently flipped variables [3]. This improvement to GSAT resulted in HSAT. McAllester, Selman and Kautz introduced Novelty and R-Novelty heuristics into the Walksat family by combining two concerns when picking a variable to flip from within a unsatisfied clause: favoring the best variable to maximize the number of satisfied clauses and avoiding flipping the most recently flipped variable in the clause to prevent the local search from repeating earlier flips [10].

Novelty and R-Novelty are among the best local search methods. However, Hoos [5] showed that they are essentially incomplete in the sense that in some situations, they can get stuck in a local basin of attraction and fail to get out. Hoos developed slightly modified procedures Novelty+ and R-Novelty+ by forcing a random walk with a fixed probability in which the variable to flip is randomly picked from a random unsatisfied clause. Novelty+ and R-Novelty+ are probabilistically approximately complete (PAC), meaning that by running them long enough, the probability of missing an existing satisfying assignment can be made arbitrarily small [5].

We note that on one hand, the variable to flip is always randomly picked in the Walksat family procedures from a unsatisfied clause in the sense the unsatisfied clause is randomly selected, and on the other hand, the choice of the variable to flip in Novelty and R-Novelty also involves some determinism: the picked variable is necessarily one of the two best variables in the unsatisfied clause.

In this paper, we propose new local search procedures in the Walksat family in two ways:

(i) We weaken the determinism in Novelty in a way that all variables in the randomly selected unsatisfied clause c can be picked to be flipped instead of limited to the two best variables in c. Concretely, we introduce diversification moves in which THE least recently flipped variable in c is picked to be flipped, resulting a new heuristic called Novelty++. Novelty++ can be considered as a reinforcement of Novelty+ in the sense that while Novelty+ makes a random walk in which all variables in c can be picked with equal probability, Novelty++ deterministically picks THE least recently flipped variable in c in a diversification step.

(ii) We weaken the randomness of the Walksat family procedures by combining GSAT with Walksat. In some precisely defined situations, the variable to flip is picked in a deterministic way as in GSAT instead of from a randomly selected unsatisfied clause. We call the new procedure G^2WSAT for Gradient-based Greedy Walksat. In the remaining situations, G^2WSAT uses Novelty++ or other Walksat family heuristics to pick the variable to flip.

This paper is organized as follows. Section 2 presents Novelty++ and compares its performance with Novelty and Novelty+. Section 3 presents G^2WSAT and compares the performance of G^2WSAT using Novelty++ with SDF [11] and UnitWalk [4], the two other effective local search procedures, as well as Novelty and Walksat. Section 4 concludes.

2 Extending Novelty with Diversification

Originally introduced in [13], Walksat differs from its predecessor GSAT essentially in that the variable to be flipped is no longer (deterministically with probability 1-p and randomly with probability p) picked among all variables but from a randomly selected unsatisfied clause. It starts with a randomly generated truth assignment. Then it repeatedly changes (flips) the assignment of a variable picked according to a heuristic, until either a satisfying assignment is found or a given maximum number of flips, *Maxsteps*, is reached. This process is repeated up to a maximum number of *Maxtries* times.

Algorithm 1: Walksat

```
Input: SAT-formula $\mathcal{F}$, $Maxtries$, $Maxsteps$, $Heuristic$
Output: A satisfying truth assignment $A$ of $\mathcal{F}$, if found
begin
    for try=1 to $Maxtries$ do
        $A$←randomly generated truth assignment;
        for flip=1 to $Maxsteps$ do
            if $A$ satisfies $\mathcal{F}$ then return $A$;
            $c$← randomly selected clause unsatisfied under $A$;
            $v$← pick a variable from $c$ according to $Heuristic$;
            $A$← $A$ with $v$ flipped;
    return “Solution not found”;
end;
```

Let x be a variable, break(x) be the number of clauses in $\mathcal{F}$ which are satisfied by the current assignment A but would be unsatisfied if x is flipped, make(x) be the number of clauses in $\mathcal{F}$ that currently are unsatisfied but would be satisfied if x is flipped. Let score(x) be the difference of make(x) and break(x) (score(x)=make(x) - break(x)). Walksat provides several heuristics to pick a variable to flip from a randomly chosen unsatisfied clause c, among which:

Walksat(p): If there are variables x in c such that break(x)=0, randomly pick one of them, otherwise with probability p, randomly pick a variable from c and with probability 1-p, randomly pick one of variables x such that break(x) is the smallest.

Novelty(p): Sort the variables x in c by score(x), breaking ties in favor of the least recently flipped variable. Consider the best and second best variable under this sort. If the best variable is not the most recently flipped one in c, then pick it. Otherwise, with probability p, pick the second best one, and with probability 1-p, pick the best variable.

R-Novelty(p): This is the same as Novelty, except in the case where the best variable is the most recently flipped one in c. In this case, pick the variable to flip among the best and second best variables according to p and the difference of score of these two variables. For more details, see [10].

We now concentrate on Novelty heuristic. Obviously, it always picks one of the two best variables according to their score that would result in the smallest (or the second smallest) total number of unsatisfied clauses. The intuition of parameter p (called noise) is that if the best variable is the most recently flipped one in c, flipping it risks to cancel a useful earlier move. To avoid this, the second best variable is picked with probability p [10].

Hoos studied the run time behavior of Novelty and found that while Novelty is very effective, it may sometimes get stuck in a loop. As an example, Hoos gave a formula $\mathcal{F}$ consisting of the 6 following clauses:

$$\begin{aligned} c_1 &: \bar{x_1} \vee x_2 \\ c_2 &: \bar{x_2} \vee x_1 \\ c_3 &: \bar{x_1} \vee \bar{x}_2 \vee \bar{y} \\ c_4 &: x_1 \vee x_2 \\ c_5 &: \bar{z_1} \vee y \\ c_6 &: \bar{z_2} \vee y \end{aligned}$$

This formula has a unique solution x_1=x_2=1, y=z_1=z_2=0. Hoos showed that if the initial assignment is x_1=x_2=y=z_1=z_2=1, Novelty never reaches the unique solution [5], because y is never flipped.

The restart mechanism in Walksat allows to remedy the situation. However in practice this mechanism critically relies on the use of good $Maxsteps$ setting which is difficult to obtain a priori. Hoos then extended Novelty by forcing a random walk with probability wp (walk probability) in which a variable in c is randomly selected. The extended Novelty is called Novelty+.

Novelty+(p, wp): With probability wp, randomly pick a variable from c (random walk), with probability 1-wp, do as Novelty.

Let us look at Hoos example once again. The reason that Novelty gets stuck in a loop is due to the fact that y is never flipped. The purpose of the random walk in Novelty+ is to make the heuristic pick y when c_3 is selected. However the random walk does so only with probability 1/3. This observation leads us to extend Novelty in the following way:

Novelty++(p, dp): With probability dp (diversification probability), pick the least recently flipped variable in c (diversification), with probability 1-dp, do as Novelty.

Obviously, the difference between Novelty+ and Novelty++ is that the random walk in Novelty+ is replaced by the diversification in Novelty++. In practice, Novelty++ is stronger than Novelty+. For instance, Novelty++ directly picks y in c_3 in the above example in a diversification step while Novelty+ may still pick x_1 or x_2 in a random walk as Novelty does.

When Novelty gets stuck in a local basin of attraction, probably there is a clause c that is unsatisfied again and again. The diversification in Novelty++ allows to flip all variables in c by turns when the search proceeds, since after the least recently variable in c is flipped, a different variable in c becomes the new least recently flipped.

So Novelty++ presumably improves (further than Novelty+) the coverage rate of Novelty as defined in [11] to measure how systematically the search explores the entire space.

Table 1 compares the performance of Novelty(p), Novelty+(p, wp) and Novelty++(p, dp) (respectively N(p), N+(p, wp) and N++(p, dp) in the table) for random 3-SAT problems and several classes of structured problems, where p is the noise parameter which is fixed to be 0.20, 0.35 and 0.50. wp and dp respectively are the random walk probability in Novelty+ and the diversification probability in Novelty++ which are fixed to be 0.01, 0.02 and 0.05.

A total of 21 local search procedures are evaluated in table 1. All these procedures share the same implementation and the same data structure from Satz [8], and simplify the input formula by satisfying the eventual unit clauses in the formula before the local search. Note that Novelty(p)≡Novelty+(p, 0)≡Novelty++(p, 0).

We generate 2000 random 500 variable and 2125 clause 3-SAT formulas and eliminate the 912 unsatisfiable ones using Satz. For larger hard random 3-SAT problems, we generate 1000 random 600 variable and 2550 clause formulas and 300 random 1000 variable and 4250 clause formulas. However no solver is available to eliminate unsatisfiable formulas of these sizes in reasonable time. So the 1000 hard random 600 variable formulas, as well as the 300 hard random 1000 variable formulas, probably contain about a half of unsatisfiable problems also used in the experimentation. $Maxsteps$ is fixed to be 10^5 for 500 variable problems, 2×10^5 for 600 variable problems and 5×10^5 for 1000 variable problems. 10^5 is the default cutoff value of the original Walksat family procedures, while 2×10^5 and 5×10^5, as well as 10^6 and 10^7 used below, are cutoff values somewhat arbitrarily fixed here.

The behavior of the local procedures for other cutoff values deserves future study.

We run Novelty, Novelty+, Novelty++ with $Maxtries = 1$ (one run) for each formula in each class. We say a formula is *solved* by a procedure if the procedure finds a solution satisfying all clauses of the formula. The number of solved formulas in a class represents the success rate of the procedure for this class. This execution is repeated 100 times to get the final averaged success rate given in the table 1.

Structured problems Flat200-479, QG, AIS, Logistics, Blockworld, all available in SATLIB[1], are also used to evaluate the performance of Novelty++ compared with Novelty and Novelty+. In order to make the comparison clearer, we eliminate unsatisfiable formulas in the QG class, and smaller formulas in the AIS, Logistics and Blockworld classes in which larger a formula, harder it is. So in our experimentation, local search procedures are run to solve the 100 satisfiable formulas in Flat200-479 class, the 10 satisfiable formulas in QG (satQG in the table) and the largest (and the hardest) formula remaining in the AIS, Logistics and Blockworld classes. $Maxsteps$ is fixed to be 10^6 to all these problems except bw_large.d for which 10^7 flips are used. As for random problems, we run each procedure with $Maxtries = 1$ for each formula in a class to get a success rate for this class and repeat the execution 100 times to get final averaged success rate given in table 1.

[1] http://www.satlib.org

Table 1. Average success rate and successful run length (the number of flips to find a solution in a successful run) of Novelty, Novelty+ and Novelty++ for random 3-SAT and structured problems

	500vars succ rate #flip	600vars succ rate #flip	1000vars succ rate #flip	Flat200 succ rate #flip	satQG succ rate #flip	ais12 succ rate #flip	logistics.d succ rate #flip	bw_large.d succ rate #flip
N(.2)	0.0358 43919	0.0137 89461	0.0020 263457	0.0772 314589	0.610 157354	0 0	0.94 314949	0.81 3819298
N+(.2, .01)	0.0620 43407	0.0257 87354	0.0073 244555	0.1329 342308	0.626 177019	0.09 600588	0.94 272877	0.74 4092116
N++(.2, .01)	*0.0752* *41308*	*0.0331* *83556*	*0.0096* *233827*	*0.1665* *326908*	*0.621* *174334*	*0.32* *517261*	*0.97* *255227*	*0.70* *3769416*
N+(.2, .02)	0.0717 42755	0.0307 84166	0.0091 230063	0.1554 331649	0.610 166960	0.15 394956	0.98 252352	0.68 4803834
N++(.2, .02)	*0.0922* *39599*	*0.0409* *81692*	*0.0121* *216712*	*0.2023* *314755*	*0.642* *168948*	*0.47* *472868*	*0.99* *257194*	0.54 *4281634*
N+(.2, .05)	0.0944 39612	0.0407 80012	0.0138 226484	0.1911 307785	0.628 176049	0.36 530276	0.97 245133	0.47 4755044
N++(.2, .05)	*0.1299* *37599*	*0.0592* *75300*	*0.0225* *218320*	*0.2937* *314195*	*0.662* *155574*	*0.78* *370399*	*0.98* *187548*	*0.13* *4551418*
N(.35)	0.1889 39923	0.0942 81522	0.0405 233992	0.3778 336305	0.684 101838	0 0	1 150960	0.01 523016
N+(.35, .01)	0.2236 38523	0.1137 78222	0.0537 225226	0.4887 329866	0.663 82270	0.08 374812	1 139387	0.01 3935403
N++(.35, .01)	*0.2525* *36874*	*0.1315* *74005*	*0.0676* *210373*	*0.5601* *299901*	*0.655* *78833*	*0.16* *513290*	*1* *146827*	*0.01* *9034913*
N+(.35, .02)	0.2411 37778	0.1250 74336	0.0626 214503	0.5224 302060	0.660 84524	0.08 284610	1 154852	0.01 7636423
N++(.35, .02)	*0.2850* *35694*	*0.1503* *71890*	*0.0805* *197205*	*0.6283* *287165*	*0.655* *84537*	*0.28* *472346*	*1* *171744*	*0.01* *4258715*
N+(.35, .05)	0.2807 36191	0.1498 72502	0.0808 197306	0.5997 290165	0.650 80302	0.24 494390	1 169413	0.01 7666983
N++(.35, .05)	*0.3727* *34111*	*0.2036* *68382*	*0.1234* *184949*	*0.7947* *252688*	*0.634* *55898*	*0.49* *512219*	*1* *212222*	*0* *0*
N(.5)	0.5185 34169	0.3118 67335	0.2375 188880	0.8585 225958	0.613 67213	0 0	0.71 434699	0 0
N+(.5, .01)	0.5360 33618	0.3262 65780	0.2561 179796	0.8877 208462	0.612 69846	0.09 597206	0.57 487620	0 0
N++(.5, .01)	*0.5617* *32575*	*0.3449* *63487*	*0.2798* *171378*	*0.9154* *189963*	*0.604* *67538*	*0.1* *505106*	*0.50* *438835*	*0* *0*
N+(.5, .02)	0.5482 33223	0.3354 65321	0.2673 177141	0.9011 199674	0.602 59886	0.07 481570	0.43 453581	0 0
N++(.5, .02)	*0.5870* *31879*	*0.3634* *61308*	*0.3048* *162701*	*0.9223* *174457*	*0.603* *69033*	*0.21* *599057*	*0.27* *382697*	*0* *0*
N+(.5, .05)	0.5708 32511	0.3520 62915	0.2949 167434	0.9142 188466	0.601 73372	0.11 459436	0.37 529073	0 0
N++(.5, .05)	*0.5919* *30953*	*0.3641* *59633*	*0.3163* *153879*	*0.9397* *175887*	*0.593* *73184*	*0.33* *473227*	*0.14* *437366*	*0* *0*

Table 1 shows that Novelty++ is consistently better than Novelty and Novelty+ in case noise is important (random 3-SAT, Flat200) and in case stagnation behavior occurs (ais12). In other words, when random walk is needed, diversification systematically does better. Novelty++ generally has a success rate 2 or 3 times larger than Novelty+ for ais12 for the same value of wp and dp. It also solves significantly more random 3-SAT and Flat200 formulas, especially when noise parameter is low (0.2 and 0.35).

Note that the success rate in table 1 can be computed in an equivalent way as follows. We consider a multi-set based on each class where every formula occurs 100 times. We run a local search procedure with $Maxtries = 1$ for every formula in the multi-set. The number of formulas solved divided by the number of elements in the multi-set is the success rate. For example, the Flat200 class has 100 formulas. The corresponding multi-set has 100*100=10000 formulas. Novelty++(0.35, 0.05) solves 7947 formulas in the multi-set. Its success rate for the class Flat200 is 0.7947. It solves 1950 (over 10000) formulas more than Novelty+(0.35, 0.05) in the multi-set.

We recall that about a half of formulas in the random 600 and 1000 variable 3-SAT classes probably might be unsatisfiable. The success rate for these two classes probably might be multiplied by 2. Consequently, the success rate difference between Novelty++ and Novelty+ for these two classes might also be multiplied by 2.

It appears in Table 1 that QG problems are not sensitive to noise nor to random walk, since the success rates for different noise parameters don't have significant difference. The only cases Novelty+ is better than Novelty++ are the logistics.d and bw_large.d problems for which Novelty(0.2) is the best. In other words, Neither random walk nor diversification is necessary for these two problems. Moreover, noise should be low to solve them.

All evaluated procedures roughly have the same time complexity per flip. Table 1 shows that when Novelty++ has better success rate, it also generally needs fewer flips to find a solution.

3 Exploiting Promising Decreasing Paths in Local Search

The behavior of GSAT before finding a solution might roughly be characterized as follows:

1. Repeatedly decrease the number of unsatisfied clauses (move down along a decreasing path) while possible;
2. Escape from a local minimum;
3. Go to 1.

One major difference between Walksat family and GSAT is that the behavior of Walksat can no longer be characterized as above, since the variable to flip is always picked from a randomly selected unsatisfied clause and may increase the number of unsatisfied clauses in non local minima. The difficulty in GSAT is how to recognize a promising decreasing path from previous moves, since moving along a decreasing path risks to simply repeat (or cancel) earlier flips, which might explain the performance of Walksat over GSAT.

However we believe that the purpose of a local search procedure always is to repeatedly decrease the number of unsatisfied clauses. Even when the procedure flips a variable such that the number of unsatisfied clauses is increased, it hopes that decreasing variables appear and can lead to a solution satisfying all clauses.

This observation suggests us that the decreasing variables resulted from a move could be better and more promising than variables in a randomly picked unsatisfied clause c, and that in this case, a local search procedure should pick one of these decreasing variables instead of a variable in c.

We now formally define promising decreasing variables and promising decreasing paths.

A variable is said decreasing if flipping it would decrease the number of unsatisfied clauses. Let x and y be variables, $x \neq y$, y is not decreasing. If it becomes decreasing after x is flipped, then we say that y is a *promising decreasing variable* after x is flipped. There may be 0, 1, or several promising decreasing variables after x is flipped. All promising decreasing variables are collected in a set.

Let y be a promising decreasing variable after some variable is flipped. If y is always decreasing after one or more other moves, it is always promising and remains in the promising decreasing variable set. Otherwise it should be removed from the set.

A *promising decreasing path* is a sequence of moves in which every move flips a promising decreasing variable.

Note that if a variable x is flipped such that the number of clauses is increased, re-flipping x would decrease the number of unsatisfied clauses, i.e., x is decreasing. However x is not a promising decreasing variable, since re-flipping x would simply cancel a previous move. The major originality of our approach is that such x is never considered when exploiting promising decreasing paths.

We want a local search procedure which, whenever there are promising decreasing variables, does as GSAT and deterministically picks the best of them to minimize the total number of unsatisfied clauses, breaking ties in favor of the least recently flipped variable as in HSAT [3]. In other cases, the procedure does as Walksat and uses a heuristic such as Novelty++ to pick the variable to flip.

For this purpose, we need a method to efficiently find all promising decreasing variables after a flip and remove old promising decreasing variables which are no longer decreasing.

A variable x is decreasing iff score(x)=make(x)-break(x) > 0. The following gradient-based approach originally introduced in [7] allows us to compute the score of every variable after a flip.

Assume that the formula $\mathcal{F}$ contains m clauses on n variables. Let c_i $(1 \leq i \leq m)$ be a clause in $\mathcal{F}$. c_i=$x_{i_1} \vee ... \vee x_{i_k} \vee \bar{x}_{i_{k+1}} \vee ... \vee \bar{x}_{i_{k+r}}$. Note that we put all positive literals in c_i before the negative ones.

We consider now all variables in c_i as integer variables taking values 0 or 1. We define:

$$\mathcal{E}_i(x_{i_1}, ..., x_{i_k}, x_{i_{k+1}}, ..., x_{i_{k+r}}) = (1 - x_{i_1})...(1 - x_{i_k})x_{i_{k+1}}...x_{i_{k+r}}$$

Obviously $\mathcal{E}_i$ has value 0 iff one of x_{i_j} $(1 \leq j \leq k)$ is assigned 1 or one of x_{i_s} $(k + 1 \leq s \leq r)$ is assigned 0. In other words, $\mathcal{E}_i$=0 iff c_i is satisfied. otherwise $\mathcal{E}_i$=1.

We now define:

$$\mathcal{E}(x_1, ..., x_n) = \sum_{i=1}^{m} \mathcal{E}_i \tag{1}$$

Given an assignment A (a point in the search space), the value of $\mathcal{E}$ is the number of unsatisfied clauses in $\mathcal{F}$. If A satisfies all clauses in $\mathcal{F}$, then $\mathcal{E} = 0$.

Since each variable appears at most once in a clause, $\mathcal{E}$ is a linear function for any variable x_f and can be written as $\mathcal{E}(x_f)$. Let v_f be the current value of x_f. $\mathcal{E}(v_f)$ stands for $\mathcal{E}$ simplified after the substitution of x_f by v_f. Taylor's equation gives us

$$\mathcal{E}(x_f) = \mathcal{E}(v_f) + (x_f - v_f)\frac{\partial \mathcal{E}(x_f)}{\partial x_f} \tag{2}$$

So $\frac{\partial \mathcal{E}(x_f)}{\partial x_f} \neq 0$ indicates the variation of $\mathcal{E}$ when x_f changes. If $\frac{\partial \mathcal{E}(x_f)}{\partial x_f} > 0$, then $\mathcal{E}(x_f)$ increases (decreases) when x_f increases (decreases). If $\frac{\partial \mathcal{E}(x_f)}{\partial x_f} < 0$, then $\mathcal{E}(x_f)$ decreases (increases) when x_f increases (decreases). So we can get all decreasing variables at a given point A by computing $\frac{\partial \mathcal{E}(A)}{\partial x_f}$ for every variable x_f.

For example, if at point A, x_1=0, x_2=1, x_3=0, x_4=1, $\frac{\partial \mathcal{E}(A)}{\partial x_1}$=2, $\frac{\partial \mathcal{E}(A)}{\partial x_2}$=2, $\frac{\partial \mathcal{E}(A)}{\partial x_3}$=-2, $\frac{\partial \mathcal{E}(A)}{\partial x_4}$=-2, then x_1 and x_4 are increasing variables ($\mathcal{E}$ will be increased by 2 if x_1 or x_4 is flipped), and x_2 and x_3 are decreasing variables ($\mathcal{E}$ will be decreased by 2 if x_2 or x_3 is flipped). Note that score(x_1)=-$\frac{\partial \mathcal{E}(A)}{\partial x_1}$=-2, score($x_2$)=$\frac{\partial \mathcal{E}(A)}{\partial x_2}$=2, score($x_3$)=-$\frac{\partial \mathcal{E}(A)}{\partial x_3}$=2, score($x_4$)=$\frac{\partial \mathcal{E}(A)}{\partial x_4}$=-2. In other words, |score(x_f)|=|$\frac{\partial \mathcal{E}(A)}{\partial x_f}$| for all x_f, but the sign may be different.

We now rewrite Taylor's equation 2 as

$$\mathcal{E}(x_1, ..., x_f, ..., x_n) = \mathcal{E}(x_1, ..., v_f, ..., x_n) + (x_f - v_f)\frac{\partial \mathcal{E}(x_1, ..., x_f, ..., x_n)}{\partial x_f} \tag{3}$$

Then for any variable $x_g \neq x_f$, we have:

$$\frac{\partial \mathcal{E}(x_1, ..., x_f, ..., x_n)}{\partial x_g} = \frac{\partial \mathcal{E}(x_1, ..., v_f, ..., x_n)}{\partial x_g} + (x_f - v_f)\sum_{i=1}^{m} \frac{\partial^2 \mathcal{E}_i}{\partial x_f \partial x_g} \tag{4}$$

We denote the assignment after t flips by A^t in local search. The value of x_j is v_j^t ($1 \leq j \leq n$). After a new flip, A^t becomes A^{t+1} in which $v_j^t = v_j^{t+1}$ except for one variable x_f=$v_f^{t+1} = 1 - v_f^t$. Assuming that we know the value of $\frac{\partial \mathcal{E}(x_j)}{\partial x_j}$ for every x_j at point A^t, equation 4 suggests us a reasonable and efficient way to compute $\frac{\partial \mathcal{E}(x_j)}{\partial x_j}$ at point A^{t+1}.

In fact, we note that $v_f = v_f^t$ at point A^t. At point A^{t+1}, $\frac{\partial \mathcal{E}(x_1, ..., x_f, ..., x_n)}{\partial x_g}$ is $\frac{\partial \mathcal{E}(A^{t+1})}{\partial x_g}$, but $\frac{\partial \mathcal{E}(x_1, ..., v_f^t, ..., x_n)}{\partial x_g}$ is equal to $\frac{\partial \mathcal{E}(A^t)}{\partial x_g}$, since A^{t+1} differs from A^t only in the value of x_f (recall that $\frac{\partial\mathcal{E}(x_1,\ldots,v_f^t,\ldots,x_n)}{\partial x_g}$ is $\frac{\partial\mathcal{E}(x_1,\ldots,x_f,\ldots,x_n)}{\partial x_g}$ with x_f replaced by v_f^t so that all variables in it have value by A^t).

Since $x_f - v_f^t = v_f^{t+1} - v_f^t = 1 - v_f^t - v_f^t$ at point A^{t+1}, equation 4 becomes:

$$\frac{\partial\mathcal{E}(A^{t+1})}{\partial x_g} = \frac{\partial\mathcal{E}(A^t)}{\partial x_g} + (1 - 2v_f^t)\sum_{i=1}^{m}\frac{\partial^2\mathcal{E}_i}{\partial x_f \partial x_g} \tag{5}$$

Note that $\sum_{i=1}^{m}\frac{\partial^2\mathcal{E}_i}{\partial x_f \partial x_g} = 0$ for all x_g not occurring in any clause containing x_f. To see this, let $c_i = x_1 \vee \bar{x}_2 \vee x_4$, $\mathcal{E}_i = (1-x_1)x_2(1-x_4)$. $\frac{\partial^2\mathcal{E}_i}{\partial x_2 \partial x_g} = 0$ for any $g \notin \{1, 4\}$ (including $g = 2$). The following properties hold:

$$\frac{\partial\mathcal{E}(A^{t+1})}{\partial x_g} = \begin{cases} \frac{\partial\mathcal{E}(A^t)}{\partial x_g}, \text{ if } x_g \text{ not occurring in any clause with } x_f \\ \dfrac{\partial\mathcal{E}(A^t)}{\partial x_g} + (1 - 2v_f^t) \displaystyle\sum_{x_f\ and\ x_g\ occur\ in\ clause\ c_i} \dfrac{\partial^2\mathcal{E}_i}{\partial x_f \partial x_g} \\ \frac{\partial\mathcal{E}(A^t)}{\partial x_f}, \text{ if } x_g = x_f \end{cases} \tag{6}$$

Equation 6 shows that when x_f is flipped, $\frac{\partial\mathcal{E}(x_1,\ldots,x_f,\ldots,x_n)}{\partial x_g}$ should be re-computed only for those x_g occurring in some clause containing x_f. These variables and corresponding clauses can be stored by a preprocessing in a list associated with x_f to speed up the calculus.

In summary, the local search procedure, which we call G^2WSAT for Gradient-based Greedy Walksat, uses equation 6 to maintain a set of promising decreasing variables and flips the best promising decreasing variable if any. Otherwise, it uses a heuristic such as Novelty++ to pick a variable to flip.

G^2WSAT is defined in algorithm 2.

Table 2 compares the success rate of G^2WSAT using Novelty++ (G^2(p, dp) in the table) with Novelty++ (N++(p, dp) in the table) on the same problems as in table 1. Novelty++(p, 0) is just Novelty(p). Recall the only difference between G^2WSAT and Novelty++ here is that G^2WSAT flips the best promising decreasing variable if any. Otherwise it is the same as Novelty++. The success rate is also computed in the same manner. The $Maxsteps$ (cutoff value for the number of flips) is also the same as in table 1 (10^5 for random 500 variable 3-SAT, 2×10^5 for 600 variables, 5×10^5 for 1000 variables; 10^7 for bw_large.d and 10^6 for other structured problems).

Due to the lack of space, we don't give the successful run lengths (#flips) of G^2WSAT and Novelty++ in table 2. G^2WSAT generally needs fewer flips than Novelty++ to find a solution. Table 3 gives some typical examples of the successful lengths of G^2WSAT which can be compared with those of Novelty++ given in table 1.

Table 2 shows that G^2WSAT is almost always better than Novelty++ except for Flat200 problems. In particular, the best success rate is always obtained by G^2WSAT, even for Flat200 problems. It is remarkable that $G^2WSAT(0.2, dp)$ improves Novelty++(0.2, dp) and raises its success rate to 100%.

Algorithm 2: G^2WSat

```
Input: SAT-formula $\mathcal{F}$, $Maxtries$,$Maxsteps$, $Heuristic$
Output: A satisfying truth assignment $A$ of $\mathcal{F}$, if found
begin
    for try=1 to $Maxtries$ do
        $A \leftarrow$ randomly generated truth assignment;
        Compute $\frac{\partial\mathcal{E}(x_1,\ldots,x_n)}{\partial x_j}$ for all $x_j$ at $A$;
        Store all decreasing variables in stack $DecVar$;
        for flip=1 to $Maxsteps$ do
            if $A$ satisfies $\mathcal{F}$ then return $A$;
            if $DecVar$ is not empty then $x \leftarrow x$ in $DecVar$ such that $|\frac{\partial\mathcal{E}(x_1,\ldots,x_n)}{\partial x}|$ is the
            largest, breaking ties in favor of the least recently flipped variable;
            else $c \leftarrow$ randomly selected clause unsatisfied under $A$;
                $x \leftarrow$ pick a variable from $c$ according to $Heuristic$;
            $A \leftarrow A$ with $x$ flipped;
            update $\frac{\partial\mathcal{E}(x_1,\ldots,x_n)}{\partial x_j}$ for all $x_j$ using equation 6;
            delete all variables which are no longer decreasing from $DecVar$;
            push all new decreasing variables into $DecVar$ which are different from $x$ and
            were not decreasing before $x$ is flipped;
    return "Solution not found";
end;
```

As in table 1, the success rate difference between G^2WSAT and Novelty++ for random 600 and 1000 variable 3-SAT problems might probably be multiplied by 2. It appears in table 2 that the success rate difference for 3-SAT increases with the noise and diversification parameters. For example, for 500 variable 3-SAT problems, the largest success rate difference is 0.0337 when the noise parameter is 0.2, it is respectively 0.0404 and 0.0625 when the noise parameter is 0.35 and 0.5. On the other hand, when the noise parameter is 0.5, the success rate difference between G^2WSAT and Novelty++ respectively is 0.0150, 0.0177, 0.0342 and 0.0625 when the diversification parameter is 0, 0.01, 0.02 and 0.05. The same phenomenon can be observed for the 600 and 1000 variable problems.

The difference of 0.0625 here for random 500 variable 3-SAT means that G^2WSAT(0.5, 0.05) solves 6807 more formulas than Novelty++(0.5, 0.05) in our experimentation. Refer to table 1, G^2WSAT(0.5, 0.05) solves 9098 more random 500 variable 3-SAT formulas than Novelty+(0.5, 0.05), representing a success rate difference of 0.0836. Considering the hardness to improve the Walksat family procedures which are already highly effective, we believe that G^2WSAT combining with Novelty++ is a significant improvement. As a comparison, the success rate difference between Novelty(0.5) and Walksat(0.5) for these formulas is 0.0704, see table 3 below.

In table 3, we compare G^2WSAT combined with Novelty++(p, dp) (G^2(p, dp) in the table) with Walksat, Novelty, UnitWalk, SDF on the same problems as in table 1 or in table 2. In addition to the success rates computed in the same manner as in table 1, we also report the average number of flips to find a solution (i.e. average length of a successful run) and the total real run time in seconds on an Athlon2000+

Table 2. Average success rate of Novelty++ and G^2WSAT using Novelty++ for random 3-SAT and structured problems

	500vars	600vars	1000vars	Flat200	satQG	ais12	logistics.d	bw_large.d
N++(.2, 0)	0.0358	0.0137	0.0020	0.0772	0.610	0	0.94	0.81
G^2*(.2, 0)*	*0.0477*	*0.0188*	*0.0034*	*0.0693*	*0.672*	*0*	*1*	*0.93*
N++(.2, .01)	0.0752	0.0331	0.0096	0.1665	0.621	0.32	0.97	0.70
G^2*(.2, .01)*	*0.0992*	*0.0416*	*0.0123*	*0.1607*	*0.679*	*0.52*	*1*	*0.89*
N++(.2, .02)	0.0922	0.0409	0.0121	0.2023	0.642	0.47	0.99	0.54
G^2*(.2, .02)*	*0.1178*	*0.0513*	*0.0160*	*0.1955*	*0.685*	*0.68*	*1*	*0.77*
N++(.2, .05)	0.1299	0.0592	0.0225	0.2937	0.662	0.78	0.98	0.13
G^2*(.2, .05)*	*0.1636*	*0.0754*	*0.0285*	*0.2701*	*0.725*	*0.88*	*1*	0.26
N++(.35, 0)	0.1889	0.0942	0.0405	0.3778	0.684	0	1	0.01
G^2*(.35, 0)*	*0.2104*	*0.1056*	*0.0438*	*0.3207*	*0.732*	*0*	*1*	*0.17*
N++(.35, .01)	0.2525	0.1315	0.0676	0.5601	0.655	0.16	1	0.01
G^2*(.35, .01)*	*0.2884*	*0.1490*	*0.0754*	*0.5083*	*0.730*	*0.25*	*1*	*0.03*
N++(.35, .02)	0.2850	0.1503	0.0805	0.6283	0.655	0.28	1	0.01
G^2*(.35, .02)*	*0.3229*	*0.1710*	*0.0904*	*0.5778*	*0.747*	*0.40*	*1*	*0.01*
N++(.35, .05)	0.3727	0.2036	0.1234	0.7947	0.634	0.49	1	0
G^2*(.35, .05)*	*0.4131*	*0.2278*	*0.1318*	*0.7226*	*0.737*	*0.59*	*1*	*0*
N++(.5, 0)	0.5185	0.3118	0.2375	0.8585	0.613	0	0.71	0
G^2*(.5, 0)*	*0.5335*	*0.3218*	*0.2372*	*0.8312*	*0.734*	*0*	*0.84*	*0*
N++(.5, .01)	0.5617	0.3449	0.2798	0.9154	0.604	0.1	0.50	0
G^2*(.5, .01)*	*0.5894*	*0.3568*	*0.2832*	*0.9051*	*0.710*	*0.13*	*0.65*	*0*
N++(.5, .02)	0.5870	0.3634	0.3048	0.9223	0.603	0.21	0.27	0
G^2*(.5, .02)*	*0.6212*	*0.3788*	*0.3127*	*0.9283*	*0.713*	*0.25*	*0.53*	*0*
N++(.5, .05)	0.5919	0.3641	0.3163	0.9397	0.593	0.33	0.27	0
G^2*(.5, .05)*	*0.6544*	*0.4018*	*0.3521*	*0.9511*	*0.699*	*0.38*	*0.34*	*0*

under Linux of each procedure to run 100 times for each class. Note that G^2WSAT uses equation 6 to incrementally update the score of each variable, which is quite different from the original Walksat procedures. To show that the gradient-based approach improves the time performance, we use original Walksat_v37 downloaded from http://www.cs.washington.edu/homes/kautz in table 3.

UnitWalk is downloaded from http://logic.pdmi.ras.ru/~arist/UnitWalk. SDF is downloaded http://www.cs.ualberta.ca/~dale/software.html.

We run UnitWalk using the following command line

UnitWalk -f $\mathcal{F}$ -p $Maxsteps$ -r 100 -sr -T 20000

where a time cutoff of 2×10^4 seconds is set to solve formula $\mathcal{F}$, and run SDF using the following command line

sdfflt -mf $Maxsteps$ -mr 1 -rep 100 -ne -f $\mathcal{F}$

to solve formula $\mathcal{F}$.

The $Maxsteps$ value is the same as in table 1 or in table 2 for different problems.

The best success rate, #flips, total run time for each class are bold-faced in table 3. Generally, a procedure with a better success rate is faster and has shorter successful runs. However, SDF uses a float-point implementation in which search steps are more expensive. So-called substitution operations are needed in UnitWalk in addition

Table 3. Experimental results for UnitWalk, SDF, G^2WSAT, Novelty and Walksat

	500vars #flips time	600vars #flips time	1000vars #flips time	Flat200 #flips time	satQG #flips time	ais12 #flips time	logistics.d #flips time	bw_large.d #flips time
SDF	0.3674 36218 56165s	0.1707 70702 115162s	0.0437 215026 210688s	**0.9859** **134555** 7057s	0.42 93207 ?	**1** **145781** 267s	1 65080 2442s	? ? ?
UnitWalk	0.4409 41429 25132s	0.2731 76443 53733s	0.2399 205646 55543s	0.8960 325342 24646s	0.5280 134808 93620s	1 232861 10374s	1 **3791** 53s	0.02 5362045 19899s
$G^2(.2, 0)$	0.0477 40950 12263s	0.0188 86223 23166s	0.0034 236392 20951s	0.0693 312826 5532s	0.672 167887 **5452s**	0 0 221s	1 133848 28s	**0.93** **3419339** **2733s**
$G^2(.2, .05)$	0.1636 35207 11519s	0.0754 70840 22970s	0.0285 198330 21147s	0.2692 293976 4813s	0.725 162965 5642s	0.88 278722 **80s**	**1** 98065 **26s**	0.26 4543846 8867s
$G^2(.5, 0)$	0.5335 33214 8771s	0.3218 66156 19963s	0.2372 183047 19151s	0.8312 240477 2297s	**0.734** 87954 6392s	0 0 227s	0.84 438108 172s	0 0 14329s
$G^2(.5, .05)$	**0.6544** **29238** **7666s**	**0.4018** **55478** **18835s**	**0.3521** **145030** **17701s**	0.9511 154994 **1264s**	0.699 **84898** 7184s	0.38 521013 186s	0.34 565058 318s	0 0 15047s
Novelty(.2)	0.036 43200 14631s	0.0140 89277 29099s	0.0023 278784 26590s	0.0734 315588 7763s	0.098 262181 27657s	0 0 409s	0.95 296197 57s	0.79 3672728 3819s
Novelty(.5)	0.5178 34225 10786	0.3123 66702 25613s	0.2371 186607 24614s	0.8544 226115 2940s	0.189 399360 36628s	0 0 427s	0.64 467981 210s	0 0 17557s
Walksat(.2)	0.1129 41111 13582s	0.0525 82424 26269s	0.0187 234638 21509s	0.1093 337845 7115s	0.064 176761 27397s	0.26 492378 343s	0.64 617464 104s	0.51 4574077 3976s
Walksat(.5)	0.4474 40169 12074s	0.2844 77768 25334s	0.2341 202990 20614s	0.8118 296044 4155s	0.04 447541 38714s	0.05 554170 391s	0.60 476361 168s	0 0 15368s

to flips. Some irregularities may also happen in some problem classes. For example, $G^2WSAT(0.2, 0)$ solves more easily qg7-13 problems in QG class for which unsuccessful runs are very time-consuming, which explains the better time performance of $G^2WSAT(0.2, 0)$ with a lower success rate.

Table 3 shows that G^2WSAT generally has better performance than Novelty using the same parameters except for Flat200 problems, for which $G^2WSAT(p, dp)$ is not better than Novelty(p) when dp=0, but $G^2WSAT(p, dp)$ is significantly better than Novelty(p) when $dp > 0$.

When noise p is 0.2, G^2WSAT(0.2, 0) generally is not better than Walksat(0.2). Again the diversification allows to remedy the situation and makes G^2WSAT(0.2, 0.05) substantially better than Walksat(0.2). When noise p is 0.5, G^2WSAT(0.5, 0) generally is substantially better than Walksat except for ais12 for which diversification is necessary and makes G^2WSAT(0.5, 0.05) significantly better than Walksat(0.5), and except for logistics.d and bw_large.d for which noise should be low.

We observe that G^2WSAT always gives the best result among all Walksat family procedures in table 3, using appropriate noise and diversification parameters. Furthermore, it seems that G^2WSAT spends less time per flip than the original Novelty and Walksat used here, thanks to the gradient-based approach. For example, G^2WSAT(0.5, 0) spends 227 seconds to make the 100×10^6 flips to solve ais12 100 times without success, while Novelty(0.2) and Novelty(0.5) respectively spend 409 and 427 seconds to do the same thing[2].

G^2WSAT(0.5, 0.05) has a success rate significantly better than UnitWalk for random 3-SAT, Flat200 and QG problems, while G^2WSAT(0.2, 0.05) is better for QG and bw_large.d problems. UnitWalk performs well for ais12 formula with a success rate 1 while the best rate of G^2WSAT for this formula is 0.88. Nevertheless, G^2WSAT is substantially faster than UnitWalk for all problems in table 3. G^2WSAT(0.2, 0.05) also has the success rate 1 for ais12 formula using 3×10^6 flips in a run and is still much faster than UnitWalk using 10^6 flips in a run.

The same observation can be made when comparing G^2WSAT and SDF. We fail to run SDF for some QG problems. When solving bw_large.d formula, we stopped SDF after 30000 seconds.

Remark. QG problems contains several unit clauses. The local search procedures compared in tables 1 and 2 all simplify the input formula by satisfying unit clauses before the local search. It seems that the Walksat and Novelty procedures used in table 3 don't contain this simplification, which might explain the success rate difference of Novelty for QG problems. On other formulas which don't contain any unit clause, our implementation of Novelty reproduces the same success rate of the original Novelty.

4 Conclusion

We have proposed two extensions of the Walksat family local search procedures. The first extension is the diversification in Novelty so that in each search step, with probability dp the least recently flipped variable in a randomly selected unsatisfied clause c is picked to be flipped, while in the remaining cases, normal Novelty heuristic is used to pick the variable to flip in c. The new heuristic is called Novelty++. The diversification allows to weaken the determinism in Novelty which always picks one of the two best variables in c. It is also stronger than the random walk in Novelty+, since it deterministically picks the least recently flipped variable in c.

The second extension is the deterministic exploitation of so-called promising decreasing variables using a gradient-based approach. The new procedure is called G^2WSAT. We have combined G^2WSAT with Novelty++ such that whenever there

[2] After our experimentation is done, we are told that the new version (version 45) of Walksat has been speeded up (by 30-50% for large problems) using an improvement to the Walksat variable flip algorithm due to Alex Fukunaga (The search behavior (the assignments explored) is totally unaffected) [2].

are promising decreasing variables, the best (according to its score) of them is picked to be flipped, otherwise Novelty++ is used to pick the variable to flip.

Experimental results on a large number of random 3-SAT and structured problems show the performance of Novelty++ compared with Novelty and Novelty+ under different parameter settings, and the performance of G^2WSAT combined with Novelty++ compared with state-of-the-art local search procedures such as Novelty, Novelty+, Walksat, UnitWalk and SDF.

Similar to other heuristics in the Walksat family, Novelty++ is sensitive to noise and diversification parameters. An adaptive noise mechanism as presented in [6] might be used to automatically adjust the parameters when the search proceeds, in order to further improve the performance of Novelty++.

References

1. S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of 3rd Annual ACM Symp. Theory of Computing*, pages 151–158, 1971.
2. A. Fukunaga. Efficient implementation of sat local search. In *Proceedings of SAT-2004 (Seventh International Conf. on Theory and Applications of Satisfiability Testing*, Vancouver, British Columbia, 2004.
3. I. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of AAAI-93*, 1993.
4. E. A. Hirsch and A. Kojevnikov. Unitwalk: A new sat solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1-4):91–111, 2005.
5. H. Hoos. On the run-time behavior of stochastic local search algorithms for sat. In *Proceedings of AAAI-99*, pages 661–666, 1999.
6. H. Hoos. An adaptive noise mechanism for walksat. In *Proceedings of AAAI-02*, pages 655–660. AAAI Press / The MIT Press, 2002.
7. W. Q. Huang and Jin Ren Chao. Solar: a learning from human algorithm for solving sat. *Science in China (Series E)*, 27(2):179–186, 1997.
8. C.M. Li and Anbulagan. Look-Ahead Versus Look-Back for Satisfiability Problems. In *Proceedings of CP-97, Third International Conference on Principles and Practice of Constraint Programming*, pages 342–356. Springer-Verlag, LNCS 1330, Shloss Hagenberg, Austria, 1997.
9. B. Mazure, L. Saïs, and É. Grégoire. Tabu Search for SAT. In *Proceedings of AAAI'97*, 1997.
10. D.A. McAllester, B. Selman, and H. Kautz. Evidence for invariant in local search. In *Proceedings of AAAI-97*, pages 321–326, 1997.
11. D. Schuurmans and F. Southey. Local search characteristics of incomplete sat procedures. *Artificial Intelligence*, 132(2):121–150, 2001.
12. B. Selman and H. Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In *Proceedings of IJCAI-93*, 1993.
13. B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of AAAI-94, 12th National Conference on Artificial Intelligence*, pages 337–343. AAAI Press, Seattle, USA, 1994.
14. B. Selman, D. Mitchell, and H. Levesque. A new Method for Solving Hard Satisfiability Problems. In *Proceedings of AAAI'92*, pages 440–446, 1992.

On Finding All Minimally Unsatisfiable Subformulas

Mark H. Liffiton and Karem A. Sakallah

Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor 48109-2122
{liffiton, karem}@eecs.umich.edu

Abstract. Much attention has been given in recent years to the problem of finding Minimally Unsatisfiable Subformulas (MUSes) of Boolean formulas. In this paper, we present a new view of the problem, strongly linking it to the maximal satisfiability problem. From this relationship, we have developed a novel technique for extracting all MUSes of a CNF formula, tightly integrating our implementation with a modern SAT solver. We also present another algorithm for finding all MUSes, developed independently but based on the same relationship. Experimental comparisons show that our approach is consistently faster than the other, and we discuss ways in which ideas from both could be combined to improve further.

1 Introduction

Many computational problems in a wide range of fields are posed as constraint satisfaction problems, often in the form of Boolean CNF formulas analyzed with satisfiability (SAT) solvers. While SAT solvers can return a short proof in the form of a satisfying assignment when a formula is satisfiable, typically no proof or explanation is given when a formula is found to be unsatisfiable. Explanations of infeasibility are often valuable, and techniques for finding them have been developed for use in these problems. Some techniques have focused on reducing the original set of constraints to produce a minimal, unsatisfiable core representing a cause of infeasibility. In this paper, we present a new approach to finding these cores, focusing on a complete method for finding all unsatisfiable cores of any given formula.

Consider an unsatisfiable CNF formula φ . A Minimally Unsatisfiable Subformula (MUS) of φ is a subset of φ 's clauses that is both unsatisfiable and minimal in the sense that all of its proper subsets are satisfiable. An MUS can be seen as an irreducible cause of the infeasibility of the original formula. φ could have multiple reasons for its infeasibility. In this case φ would contain multiple MUSes, and fixing any single MUS may not make φ satisfiable. As long as any MUS is present in the formula, it will remain infeasible. In many applications, it is valuable to find the set of all MUSes, because diagnosing infeasibility is hard, if not impossible, without a complete view of its causes. Additionally, an algorithm that finds all MUSes provides a basis for approximations and techniques that find multiple, though not all, MUSes.

Many methods for finding MUSes have been developed in recent years, both for Boolean satisfiability problems and for other types of constraints. Most techniques find

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 173 – 186, 2005.

a single unsatisfiable subformula (US), often not guaranteeing it to be minimal. For example, AMUSE [10], Bruni & Sassano's algorithm [3,4], and zCore [13] all use information from a SAT solver's resolution procedure to find a single US, but none guarantee its minimality. For these, a "Minimal Unsatisfiability Prover" [7] can be used to minimize the US into an MUS. Chinneck and Dravnieks [5] studied MUSes in the domain of linear and integer programs, calling them Irreducible Infeasible Subsets. Their algorithms return multiple, but not all, MUSes.

Recently, we have developed a sound and complete technique for finding all MUSes of a CNF formula [9], based on a strong relationship between maximal satisfiability and minimal unsatisfiability. The relationship and algorithms derived from it also hold for any other type of constraint with a strict definition of satisfiability (i.e., they do not apply directly to "soft" constraints). Independently, Bailey and Stuckey [1] have also noted this relationship and developed an implementation for a type of constraint used for type-error debugging in software verification.

While both our work and theirs are based on the same underlying concept, the algorithms we developed to exploit it differ greatly. In this paper, we present a comparison of our two approaches, having implemented their algorithm for Boolean constraints. We show that, for the case of Boolean constraints, our algorithm outperforms theirs by one to two orders of magnitude. We further assess the differences between the two approaches and discuss the strengths and weaknesses of both. (In [1], the authors compare their algorithm to the best known previous method for finding all MUSes in [2]. They show that their algorithm is superior to that in [2], so we have not included that method in our comparison.)

The rest of this paper is organized as follows. Section 2 lays the foundation by describing the relationship between maximal satisfiability and minimal unsatisfiability that is the basis for both algorithms. In Section 3, we present our algorithm, and we present Bailey and Stuckey's algorithm in Section 4. Section 5 contains the experimental results comparing the two algorithms, with discussion and analysis in Section 6. Finally, Section 7 contains conclusions and ideas for future work.

2 Maximal Satisfiability and Minimal Unsatisfiability

Both our technique and Bailey and Stuckey's algorithm are based on a strong relationship between maximal satisfiability and minimal unsatisfiability. The Maximum Satisfiability problem (Max-SAT) is an optimization problem on a CNF formula φ in which the goal is to find an assignment to the variables of φ that maximizes the number of satisfied clauses. In other words, Max-SAT yields a satisfiable subset of φ 's clauses with maximum cardinality. For example, the formula

$$\varphi = (x_1) \wedge (\neg x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2)$$

has a Max-SAT solution with three satisfied clauses:

$$\{(\neg x_1), (\neg x_1 \vee x_2), (\neg x_2)\} .$$

Whereas the Max-SAT problem has been defined with the cardinality of a subset of clauses as the optimization goal, the problem can be relaxed to have *inaugmentability* as the goal instead. For this, we define a new problem, *Maximally Satisfiable Subset* (MSS). Each MSS of a formula φ is a subset of the clauses in φ that is satisfiable and inaugmentable; adding any of the other clauses in φ to an MSS will render it unsatisfiable. The definition of the set of MSSes of a formula φ follows, with the set of MUSes defined similarly for comparison:

$$\text{MSSes}(\varphi) = \begin{Bmatrix} m \subseteq \varphi: \; m \text{ is satisfiable, and} \\ \forall c \in (\varphi - m), \; m \cup \{c\} \text{ is unsatisfiable} \end{Bmatrix}$$

$$\text{MUSes}(\varphi) = \begin{Bmatrix} m \subseteq \varphi, \; m \text{ is unsatisfiable, and} \\ \forall c \in m, \; m - \{c\} \text{ is satisfiable} \end{Bmatrix}$$

Notice that MSSes(φ) and MUSes(φ) are essentially duals of one another! An MSS is satisfiable and cannot be made larger, and an MUS is *un*satisfiable and cannot be made *smaller*. Their relationship is more than cosmetic; the complete set of one type (i.e., all MSSes or all MUSes) is actually an implicit encoding of the other.

The distinction between Max-SAT and MSS is subtle. Any Max-SAT solution is also an MSS; the maximal cardinality of the Max-SAT solution entails the inaugmentability required to be an MSS. MSSes, however, may be of different sizes, and not all will necessarily have maximum cardinality, as shown by the earlier example formula. One MSS of that formula is $\{(\neg x_1), (\neg x_1 \vee x_2), (\neg x_2)\}$, which corresponds to the Max-SAT solution. $\{(x_1), (\neg x_2)\}$ is another MSS (adding either of the other two constraints would make it unsatisfiable), though it is smaller.

Now consider (x_1), the clause not included in the Max-SAT solution to our earlier example. Removing this clause from φ makes it satisfiable. In general, given any MSS of an infeasible formula, the clauses *not* in that MSS describe an irreducible "fix" to the formula in the sense that removing them will correct the infeasibility. Therefore, we define a "CoMSS" as the complement of an MSS, and the set of CoMSSes is simply:

$$\text{CoMSSes}(\varphi) = \{m \subseteq \varphi: \; (\varphi - m) \in \text{MSSes}(\varphi)\}$$

This complementary view of MSSes provides the real link to MUSes. Note that the presence of any MUS in a CNF formula makes that formula unsatisfiable. Therefore, to correct the infeasibility by removing constraints, at least one constraint from every MUS must be removed in order to "neutralize" all of the MUSes. A CoMSS of a formula φ is a set of constraints whose removal renders φ satisfiable, thus every CoMSS must contain at least one clause from each MUS of φ.

This relationship between a CoMSS and the set of MUSes of a formula can be described as a solution to a set covering problem. Specifically, a CoMSS is a *hitting set* of the set of MUSes. A hitting set of a collection of sets is a set that contains at least one element from each set in the collection. In this case, the collection is the set of MUSes, and the CoMSS is a set of clauses that contains at least one clause from every MUS. A CoMSS is a hitting set of the MUSes with the additional restriction that it cannot be any smaller without losing its defining property: it is an *irreducible hitting set*. Figure 1 shows the example formula from above along with its MSSes, CoMSSes, and MUSes to illustrate the relationship. Notice how each CoMSS is a hitting set of the MUSes.

$$\varphi = \overset{A}{(x_1)} \wedge \overset{B}{(\neg x_1)} \wedge \overset{C}{(\neg x_1 \vee x_2)} \wedge \overset{D}{(\neg x_2)}$$

MSSes(φ)	**CoMSSes(φ)**
{B,C,D}	{A}
{A,D}	{B,C}
{A,C}	{B,D}

MUSes(φ)
{A,B}
{A,C,D}

Fig. 1. Example formula with its MSSes, CoMSSes, and MUSes

In line with the duality noted earlier, every MUS of a formula is an irreducible hitting set of the CoMSSes of that formula (which also can be seen in Figure 1). The intersection of any CoMSS of a formula with any MUS of that formula must be non-empty. The duality between CoMSSes and MUSes comes from this commutative relationship between them. The relationship also gives us a way to compute all MUSes of a formula, and it is the basis for the two algorithms compared in this paper.

3 Finding All MUSes

In our algorithm for finding all MUSes of a formula φ, we decompose the process into two steps: 1) Finding the complete set of CoMSSes of φ, and 2) Extracting MUSes from the set of CoMSSes. This decomposition follows naturally from the relationship described in the previous section. And it has a nice property in that each step is independent of the other, which allows different algorithms to be used for each step without affecting the other.

3.1 Finding CoMSSes(φ)

Every CoMSS is the set of clauses *not* included in some maximally satisfiable subformula. To find MSSes, we incrementally solve a Max-SAT problem, removing solutions as they are found to continue the search until all have been found. Figure 2 provides a pseudocode outline of the procedure.

Our algorithm is tightly integrated with and takes advantage of a modern SAT solver. We find maximally satisfiable subsets of constraints by giving the solver the ability to enable and disable constraints and check for the satisfiability of the enabled constraints all within a single search tree. Every clause $C_i = x_i^1 \vee \ldots \vee x_i^m$ in φ is augmented with a negated *clause selector variable* y_i to give $C_i' = \neg y_i \vee x_i^1 \vee \ldots \vee x_i^m$ in a new formula φ'. This is accomplished by **AddYVars** in the pseudocode. While solving φ', assigning a certain y_i FALSE has the effect of disabling or removing C_i from the set of constraints, as the augmented clause is satisfied by the assignment to y_i. Conversely, assigning y_i TRUE enables the original clause. An MSS can be obtained by finding a satisfying assignment with a minimal number of y_i variables assigned FALSE, which ensures that as few constraints as possible are disabled. The clauses left unsatisfied, which are a CoMSS, are indicated by the set of y_i variables assigned FALSE.

```
CoMSSes(formula)
    // formula is a CNF instance
    formula ← AddYVars(formula)
    bound ← 1
    CoMSSes ← ∅
    While (Sat(formula))
        boundedFormula ← AddAtMost(formula, bound)
        Repeat
            newCoMSS ← IncrementalSat(boundedFormula)
            If (newCoMSS = ∅)
                End Repeat
            CoMSSes ← CoMSSes ∪ {newCoMSS}
            boundedFormula ← AddBlocking(boundedFormula, newCoMSS)
            formula ← AddBlocking(formula, newCoMSS)
        bound ← bound+1
    Return CoMSSes
```

Fig. 2. Pseudocode for finding all CoMSSes of a formula

Instead of solving an optimization problem for every solution, however, we utilize a sliding objective approach that allows us to use a more efficient incremental search. We set a bound on the number of y_i that may be assigned FALSE using an AtMost bound. Given a set of literals $\{l_1, l_2, \ldots, l_n\}$ and a positive integer k, an AtMost bound is defined as

$$\text{AtMost}\left(\{l_1, l_2, \ldots, l_n\}, k\right) \equiv \sum_{i=1}^{n} \text{assign}\left(l_i\right) \leq k$$

where assign(l_i) is 1 if l_i is assigned TRUE and 0 otherwise. In practice, an efficient implementation of this constraint propagates the negation of each of the remaining literals in its set once k of them have been assigned TRUE. Because we only use one such constraint at a time, the same effect could easily be created by modifying any standard SAT solver to "watch" the assignments to the variables involved in the constraint and to force the remaining assignments as necessary once the bound has been reached.

In this application, we add a constraint of the form AtMost($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, k$) to place a bound on the number of disabled constraints. For each value of the bound, starting at 1 and incrementing by 1, we exhaustively search for all satisfiable assignments to the augmented formula φ', which will find all CoMSSes the size of the bound. Within this search, the incremental SAT solver can utilize learned clauses and dynamic variable ordering heuristics to full effect.

When one solution is found, the corresponding CoMSS is recorded, and the search continues incrementally after adding a blocking clause that forces out that solution. The blocking clause is a disjunction of the y variables for the clauses in that CoMSS. For example, if the solution contains $y_2 = y_4 = y_7 = F$, indicating that $\{C_2, C_4, C_7\}$ is a CoMSS, then we add a blocking clause: $y_2 \vee y_4 \vee y_7$. This excludes the CoMSS, and any supersets of it, from future solutions. Finding CoMSSes in order of increasing size

```
ExtractMUS(CoMSSes)
MUS $\leftarrow \varnothing$
While (CoMSSes $\neq \varnothing$)
    curCoMSS $\leftarrow$ Pop(CoMSSes)
    newClause $\leftarrow$ Pop(curCoMSS)
    MUS $\leftarrow$ MUS $\cup$ {newClause}
    // Remove all clauses in curCoMSS from all remaining CoMSSes
    For Each clause $\in$ curCoMSS
        For Each testCoMSS $\in$ CoMSSes
            If (clause $\in$ testCoMSS)
                testCoMSS $\leftarrow$ testCoMSS - {clause}
    // Remove any CoMSSes containing newClause
    For Each testCoMSS $\in$ CoMSSes
        If (newClause $\in$ testCoMSS)
            CoMSSes $\leftarrow$ CoMSSes - {testCoMSS}
Return MUS
```

Fig. 3. Pseudocode for extracting a single MUS from a set of CoMSSes

(i.e., MSSes in order of decreasing size) and excluding supersets from future solutions ensures that only irreducible CoMSSes are found.

As long as we are *adding* constraints (the blocking clauses), we can use an incremental search. For each search with a particular AtMost bound, every new solution is removed with a blocking clause and the search continues until no further solutions exist for the current bound. Incrementing the bound, however, relaxes a constraint on the system, so the search must start over with a new copy of the formula, augmented with all blocking clauses created thus far.

Before beginning the search with the next bound, the algorithm checks that φ' augmented with all collected blocking clauses is still satisfiable without any bound on the y_i variables. If there is no satisfying assignment, even with no restrictions on the y_i variables, the entire set of CoMSSes has been found, and the algorithm terminates.

3.2 Obtaining MUS (φ)

The set of CoMSSes implicitly encodes the entire set of MUSes of a formula, and information can be extracted from it in a variety of ways. Here, we focus on methods for extracting MUSes, though it is likely that other useful data can be obtained by analyzing the set as well.

Extracting a Single MUS in Polynomial Time

Every MUS of a formula φ is an irreducible hitting set of the CoMSSes of φ. Although MINIMAL-HITTING-SET is an NP-Hard problem [8], irreducibility is a less strict requirement than minimal cardinality. Along with the fact that no CoMSS is a subset of any other, this allows us to find an irreducible hitting set for the set of CoMSSes in polynomial time. We can generate an MUS by a greedy, iterative construction, with no search necessary. Figure 3 outlines the construction in pseudocode.

Intuitively, we want to generate a set of clauses with at least one clause from each CoMSS, such that every clause is an essential element of the set. By "essential" we mean that removing a clause will leave at least one CoMSS unrepresented in the generated MUS; this enforces the irreducibility requirement.

The algorithm works by sequentially adding clauses to a forming MUS. In the main loop, a clause is selected for inclusion in the MUS, the working set of CoMSSes is altered to force that clause to be essential, and the process iterates with the altered set of CoMSSes. The first two lines in the loop choose a CoMSS and a clause from that CoMSS (the choices can be arbitrary). The clause is added to the MUS. Then, all of the other clauses in the chosen CoMSS are removed from the remaining problem. This prevents any of those clauses from being added to the MUS in later iterations, which could make the chosen clause non-essential. Next, any CoMSSes containing the chosen clause are removed, because they are now represented in the MUS. After these modifications are made, the algorithm iterates with the resulting set of CoMSSes. When no more CoMSSes remain, the constructed set of clauses is a complete MUS.

Extracting All MUSes

Finding all MUSes involves searching for all irreducible hitting sets of the set of CoMSSes. In general, this may be impractical due to the possibly exponential number of MUSes, but in many cases the result is tractable.

Our algorithm for extracting the complete set of MUSes from the CoMSSes uses the general form of the algorithm in Figure 3. The order in which CoMSSes and clauses are selected (the choices made in the first two lines of the while loop) determines the particular MUS created by the algorithm; therefore, by branching on these two decisions, all possible MUSes can be generated. We implemented this with a recursive algorithm that takes as input the remaining set of CoMSSes (initially the entire set) and the MUS under construction in the given branch of the recursion (initially the empty set). The branching is not ideal, and many duplicate branches are encountered in practice. We employ ordering heuristics to prune as many duplicate branches as possible without missing any MUSes.

4 Dualize and Advance

The algorithm developed by Bailey and Stuckey in [1] was implemented to find all minimally unsatisfiable subsets of systems of Herbrand constraints, used for type-error debugging of Haskell programs. It exploits the same relationship between maximal satisfiability and minimal unsatisfiability as ours, however, and so it can be applied to any type of constraint as well. We present an overview of their algorithm here.

They call the algorithm "Dualize and Advance" (DAA), as it interleaves the use of the hitting-set duality with the search for what we call CoMSSes. Whereas our technique finds all CoMSSes before finding hitting sets of them, DAA computes hitting sets on a partial set of CoMSSes after finding each CoMSS — it outputs any MUSes found at that stage and also uses the results to direct the search for the next CoMSS. The full DAA algorithm is presented in pseudocode in Figure 4.

```
DAA(Constraints)
MUSes $\leftarrow \emptyset$
CoMSSes $\leftarrow \emptyset$
Seed $\leftarrow \emptyset$
Repeat
    MSS $\leftarrow$ Grow(Seed, Constraints)
    CoMSSes $\leftarrow$ CoMSSes $\cup$ {Constraints - MSS}
    PotentialMUSes $\leftarrow$ ExpandHittingSets(MUSes, {Constraints - MSS})
    Seed $\leftarrow \emptyset$
    For Each S $\in$ (PotentialMUSes - MUSes)
        If (Sat(S))
            Seed $\leftarrow$ S
            Break
        Else
            MUSes $\leftarrow$ MUSes $\cup$ {S}
Until (Seed = $\emptyset$)
Return MUSes

Grow(S, Constraints)
For Each c $\in$ (Constraints - S)
    If (IncrementalSat(S $\cup$ {c}))
        S $\leftarrow$ S $\cup$ {c}
Return S

ExpandHittingSets(MUSes, CoMSS)
If MUSes = $\emptyset$
    For Each c $\in$ CoMSS
        MUSes $\leftarrow$ MUSes $\cup$ {c}
Else
    // Compute cross product
    newMUSes $\leftarrow \emptyset$
    For Each MUS $\in$ MUSes
        For Each c $\in$ CoMSS
            newMUSes $\leftarrow$ newMUSes $\cup$ {(MUS $\cup$ {c})}
    // Perform minimization
    newMUSes $\leftarrow$ Sort(newMUSes)// in order of increasing cardinality
    MUSes $\leftarrow \emptyset$
    For Each testMUS $\in$ newMUSes
        If ($\forall$ m $\in$ MUSes. m $\not\subseteq$ testMUS)
            MUSes $\leftarrow$ MUSes $\cup$ {testMUS}
Return MUSes
```

Fig. 4. Dualize and Advance pseudocode

DAA finds MSSes in a straightforward manner by “growing” them. Given a seed in the form of a satisfiable set of constraints (which is empty in the first iteration), an MSS is constructed by attempting to add each of the problem's remaining constraints to the seed, only keeping those which do not create a conflict. After going through all possible constraints, this process will have collected a maximally satisfiable subset of the constraints, because those excluded were left out specifically because they would make it unsatisfiable. This is all contained in the **Grow** subroutine.

When an MSS is found in this manner, the complement is added to the growing set of CoMSSes. At this point, the hitting sets of the CoMSSes are computed to potentially output MUSes and/or create a new seed. Each minimal hitting set that is unsatisfiable is an MUS (they are all guaranteed to be unsatisfiable only if the complete set of CoMSSes is used), and if one is found that is satisfiable, then this set is used as a seed for the next iteration. The seed created in this way is guaranteed to not intersect with any of the MSSes found thus far because it was created from the CoMSSes. For every MSS found previously, the seed will contain at least one constraint not in that MSS. Thus, the MSS grown from the seed is guaranteed to be new.

Bailey and Stuckey mention that there are many ways to compute minimal hitting sets, noting that the problem is equivalent to the hypergraph transversal problem. For their implementation, they chose a simple method that “is simple to implement and behaves reasonably efficiently.” They compute the hitting sets of a set *G* by ordering the sets in *G*, then computing partial cross products of those sets, minimizing the results at each step. In the case of the DAA algorithm, the process can be made incremental. At each iteration, only a single set is added to the set of CoMSSes; by remembering the hitting sets computed in the last iteration, the hitting sets for the current iteration can be computed by taking the cross products of the new CoMSS with each of the hitting sets from the previous iteration. Their paper described the process, including the minimization, in mathematical terms, which we interpreted and implemented algorithmically as shown in Figure 4.

Bailey and Stuckey also discuss an optimization to their algorithm in the form of a heuristic for the order of adding constraints in the **Grow** subroutine. They aim to collect CoMSSes in increasing order of size to optimize the partial hitting set calculations. Using the constraint interaction graph, they estimate which constraints are most likely to cause unsatisfiability, ordering the constraints to choose these later in the **Grow** subroutine. In their results, the heuristic only decreased the runtime by at most half, and in some cases its use resulted in longer runtimes. Due to this and the lack of details describing it in the paper, we did not implement this heuristic in our version of DAA.

5 Results

We evaluated both our own technique and our implementation of Bailey and Stuckey's DAA algorithm using a large set of unsatisfiable CNF benchmarks from automotive product configuration [11,12]. Each benchmark encodes a set of available configurations for a product, along with constraints enforcing a specific property to be checked. We observed that the encodings were not “tight,” in that they contained numerous duplicate clauses. Duplicate clauses can yield a combinatorial explosion of MUSes, so they were removed before gathering data. There are a total of 84 benchmarks in the set, each with around 1500-1800 variables and 4000-8000 clauses. This set of benchmarks was chosen because of the range of results it provides. Though all of the instances were generated in the same manner and have the same general size, the number and size of CoMSSes and MUSes in each instance vary widely. Some have a single MUS, while others have millions; runtimes can range from less than a millisecond to days or longer.

All of the algorithms were implemented in C++. Both our algorithm for finding CoMSSes and our implementation of DAA used MiniSAT [6] as a framework for constraint solving. MiniSAT is primarily a SAT solver, but it can be extended to handle other types of constraints as well. This made it possible to integrate AtMost constraints alongside the standard Boolean CNF clauses for our CoMSSes algorithm. The data were collected in Linux on a PC with a 2.2GHz Opteron processor and 8GB of RAM.

We ran every instance against both algorithms, with a 600 second timeout for both. Of the 84 benchmarks, all MUSes were found for 31 of them within the timeout by at least one of the two algorithms. The results for these 31 are presented in Table 1. The first column lists the benchmark name, and columns 2 and 3 give the size of each benchmark with the number of variables and clauses, respectively. The following three columns list the time in seconds our algorithm spent in finding the set of CoMSSes, the time spent on the set of MUSes, and the total time as the sum of both. The seventh column lists the runtime in seconds of the DAA algorithm for finding the complete set of MUSes. The “Ratio” column provides the ratio of the runtime of DAA to that of our algorithm (column 7 divided by column 6). Finally, the last two columns list the number of CoMSSes and the number of MUSes in each benchmark.

Of the 31 benchmarks for which at least one algorithm completed, ours finished all 31 while DAA reached the 600 second timeout for 6. Additionally, our algorithm is consistently faster than DAA, usually by about one to two orders of magnitude. The benchmarks which DAA was not able to complete all had more than 10,000 MUSes, indicating that calculating sets of potential MUSes at each stage was taking most of the time. Our algorithm is likewise affected by benchmarks with large numbers of MUSes, but because the set of MUSes is only calculated once, the impact is much smaller.

We should also note that of the 53 benchmarks that neither algorithm completed, ours was able to find the complete set of CoMSSes for 18. These instances all timed out in the MUS extraction stage. For each of the 18 instances, many MUSes were generated (up to 4.5 million) before the 600 second timeout was reached. This illustrates how the problem can be made intractable by the sheer number of MUSes. In one instance, C170_FR_SZ_95, our algorithm found the complete set of CoMSSes in just 0.34 seconds, and it had generated more than 1.5 million MUSes by the time the 600 second timeout was reached.

6 Analysis

The performance numbers paint a clear picture that our algorithm is faster than DAA for Boolean constraints. However, the performance of each algorithm is dependent on

Table 1. Performance Results

Benchmark			Our Algorithm			DAA	Ratio	Solutions	
Name	#V	#C	CoMSSes (sec)	MUSes (sec)	Sum (sec)	DAA (sec)	$\frac{\text{DAA}}{\text{Sum}}$	# CoMSSes	# MUSes
C208_FC_SZ_128	1513	4469	0.06	0.00	0.06	11.1	201.8	32	1
C208_FC_SZ_127	1513	4469	0.06	0.00	0.06	12.0	206.9	34	1
C208_FA_SZ_121	1516	4247	0.07	0.00	0.07	9.9	135.3	32	2
C208_FA_SZ_120	1516	4247	0.07	0.00	0.07	10.5	141.9	34	2
C202_FS_SZ_122	1556	5385	0.09	0.00	0.09	16.5	189.7	33	1
C170_FR_SZ_92	1528	4195	0.13	0.00	0.13	38.2	293.8	131	1
C202_FW_SZ_124	1561	7435	0.15	0.00	0.15	27.6	190.3	33	1
C210_FS_SZ_130	1607	4894	0.18	0.00	0.18	11.1	61.0	31	1
C210_FS_SZ_129	1607	4894	0.19	0.00	0.19	12.1	63.4	33	1
C210_FW_SZ_136	1628	6384	0.25	0.00	0.25	17.0	67.2	31	1
C202_FW_SZ_123	1561	7437	0.26	0.00	0.26	25.0	96.9	38	4
C210_FW_SZ_135	1628	6384	0.26	0.00	0.26	18.3	70.7	33	1
C168_FW_UT_852	1804	6756	0.45	0.00	0.45	15.1	33.5	30	102
C168_FW_UT_851	1804	6758	0.45	0.00	0.45	15.2	33.6	30	102
C168_FW_UT_854	1804	6753	0.45	0.00	0.45	15.2	33.6	30	102
C168_FW_UT_855	1804	6752	0.47	0.00	0.47	15.3	32.3	30	102
C220_FV_RZ_14	1530	4013	0.57	0.00	0.57	4.3	7.7	20	80
C208_FA_RZ_64	1516	4246	0.61	0.00	0.61	48.0	78.8	212	1
C220_FV_SZ_121	1530	4035	0.65	0.00	0.65	25.3	38.9	102	9
C208_FC_RZ_70	1513	4468	0.67	0.00	0.67	53.9	80.9	212	1
C202_FS_SZ_121	1556	5387	0.83	0.00	0.83	9.5	11.4	24	4
C202_FW_RZ_57	1561	7434	1.11	0.00	1.11	137.0	123.3	213	1
C208_FA_SZ_87	1516	4255	0.46	1.41	1.87	>600	>320.5	139	12884
C170_FR_RZ_32	1528	4067	0.64	1.96	2.60	>600	>230.4	242	32768
C220_FV_RZ_13	1530	4014	1.22	2.34	3.56	47.7	13.4	76	6772
C208_FA_UT_3254	1805	6153	1.63	8.94	10.57	>600	>56.8	155	17408
C208_FA_UT_3255	1805	6156	1.68	18.40	20.08	>600	>29.9	155	52736
C210_FS_RZ_40	1607	4891	0.44	30.10	30.54	73.7	2.4	212	15
C210_FW_RZ_59	1628	6381	0.56	30.40	30.96	114.0	3.7	212	15
C220_FV_RZ_12	1530	4017	1.23	65.20	66.43	>600	>9.0	150	80272
C220_FV_SZ_65	1530	4014	2.05	65.80	67.85	>600	>8.8	198	103442

a number of factors, and in this section we discuss the details behind the performance, comparing the strengths and weaknesses of each algorithm.

One difference contributing greatly to the performance of our algorithm is its tight integration with a modern SAT solver. By formulating the problem with clause-selector variables, we let the SAT solver handle the search for MSSes itself. Additionally, by finding multiple MSSes (of a single size) within a single search tree, we immediately take advantage of all of the features of modern SAT solvers, especially learned clauses. While DAA can use an incremental search within the **Grow** subroutine, it must restart the search after any added constraint makes the growing MSS unsatisfiable. It also restarts with a new search tree for every MSS, as compared to our algorithm which only restarts the search after all MSSes of a particular size have been found.

Note that while our approach is more heavily integrated with a SAT solver, it is still fairly independent of the particular solver itself. It can be implemented with any SAT solver that provides an incremental solving interface, allows the addition of constraints mid-search, and supports the AtMost constraint. (While the last requirement is not standard, its implementation in MiniSAT is quite simple, and as noted earlier, the effect can be obtained by modifying other SAT solvers with little difficulty.) The DAA algorithm simply calls a standard solver as a subroutine, making it even simpler to implement with different solvers.

The strength of our algorithm's integration with the solver is also a drawback, in that it makes it less immediately applicable to other constraint types. We rely on the ability to encode constraint enabling and disabling within the syntax of the constraints themselves. This is easy to do with Boolean disjunctions, but other types of constraints may not be as suitable. DAA, on the other hand, can immediately be implemented using a solver of any type of constraint.

Another large difference between our algorithm and DAA is the distinction between our serial, two-phase algorithm and DAA's interleaved approach. Obtaining MUSes before computing the entire set of CoMSSes is beneficial in applications that do not require the complete set of CoMSSes nor all MUSes because it can provide results sooner. The interleaved approach could easily be adopted in our algorithm. Hitting sets of the partial set of CoMSSes could be calculated after every CoMSS is found, between stages of the incremental search (when incrementing the bound on CoMSS size), or at any desired interval. This could add a great deal of overhead, however, especially if every potential MUS had to be checked for unsatisfiability (as opposed to aborting after one set is found to be satisfiable, as DAA does to use that set as the next seed). This seems to be the case for DAA, as the instance for which it had its fastest runtime also had the fewest CoMSSes, and thus the fewest incremental hitting-set calculations. Though our algorithm could be interleaved to potentially gain efficiency, DAA could not be "de-interleaved," as it depends on the set of potential MUSes to provide the seed for the next iteration and to determine when it has found all CoMSSes.

The process of "growing" an MSS from a seed has potential application within our algorithm as well. Recall that we restart the search for MSSes after exhausting the search space for each bound on the CoMSS size. Each restart throws away valuable learned clauses. We could relax the AtMost constraints to search for CoMSSes of 2, 3, or more sizes at one time. For example, by starting with an AtMost bound of 3 on the clause selector variables, the algorithm would find CoMSSes of size 1, 2, and 3 within one search tree. There would be no guarantee that those of size 2 or 3 are irreducible, however, so the **Grow** subroutine could be used to maximize the corresponding satisfiable subset (thus minimizing the CoMSS). When that search tree is exhausted, the bound could be increased by 3, finding CoMSSes of size 4, 5, and 6 in the next iteration. In general, the range of sizes covered by each iteration could be extended at the cost of increased overhead from calls to **Grow**. The impact of this tradeoff is unclear, and it should be investigated to determine an optimal range.

Finally, the constraint-graph heuristic used in DAA to guide it towards smaller CoMSSes could be adapted for our algorithm. For example, the ranking of constraints generated by the heuristic could be implemented as an influence on the variable ordering of the clause-selector variables. In general, heuristics specific to finding CoMSSes and relevant to the constraints themselves (as opposed to the formula's variables) could be effective for both algorithms.

7 Conclusion and Future Work

We have presented a relationship between maximal satisfiability and minimal unsatisfiability that can be used to find all Minimally Unsatisfiable Subformulas of a Boolean CNF formula. We experimentally compared two methods that exploit this relationship, our own algorithm and DAA, developed by Bailey and Stuckey [1]. The results show that ours is about one to two orders of magnitude faster. We discussed the relative strengths and weaknesses of each approach, examining the causes of the performance difference as well as practical implementation details.

There are many possible directions for future improvement. Though ours is the fastest known algorithm for finding all MUSes of a Boolean CNF formula, it does not scale nearly as well as modern SAT solvers. While this is unavoidable due to the much greater complexity of the problem, work should be done to make it more efficient and practically useful. We discussed some ways in which our algorithm could be combined with ideas from Bailey and Stuckey's approach which might lead to increased performance. Additionally, performance may be increased by relaxing optimality constraints.

Finally, we believe that the relationship on which both of the algorithms in this paper are based should be explored further. The relationship between MUSes, CoMSSes, and the general idea of constraint conflicts could yield further algorithms and practical applications.

Acknowledgement

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).

References

[1] J. Bailey and P. J. Stuckey. “Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization.” In *Proc. of the 7th International Symposium on Practical Aspects of Declarative Languages* (PADL05), volume 3350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[2] M. de la Banda, P. Stuckey, and J. Wazny. “Finding All Minimal Unsatisfiable Subsets.” In *Proc. of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming* (PPDP 2003), pages 32-43, 2003.

[3] R. Bruni and A. Sassano. “Restoring Satisfiability or Maintaining Unsatisfiability by Finding Small Unsatisfiable Subformulae.” *Electronic Notes in Discrete Mathematics*, vol. 9, 2001.

[4] R. Bruni. “Approximating Minimal Unsatisfiable Subformulae by Means of Adaptive Core Search.” *Discrete Applied Mathematics*, vol. 130(2), pages 85–100, 2003.

[5] J.W. Chinneck and E.W. Dravnieks, “Locating Minimal Infeasible Constraint Sets in Linear Programs.” *ORSA Journal on Computing*, Vol. 3, No. 2, pp. 157-168, 1991.

[6] N. Eén and N. Sörensson. “An Extensible SAT-solver.” In *Sixth International Conference on Theory and Applications of Satisfiability Testing* (SAT03), 2003.

[7] J. Huang. “MUP: A Minimal Unsatisfiability Prover.” In *Proc. of the Tenth Asia and South Pacific Design Automation Conference* (ASP-DAC), January 2005.

[8] R. M. Karp. “Reducibility Among Combinatorial Problems.” In *Proc. of a Symposium on the Complexity of Computer Computations*, pages 85-103, 1972.

[9] M. Liffiton, Z. Andraus, and K. Sakallah. “From Max-SAT to Min-UNSAT: Insights and Applications.” *Technical Report CSE-TR-506-05*, University of Michigan, 2005.

[10] Y. Oh, M. Mneimneh, Z. Andraus, K. Sakallah, and I. L. Markov. “AMUSE: A Minimally-Unsatisfiable Subformula Extractor.” In *Proc. of the 41st Annual Conference on Design Automation*, pages 518–523, ACM Press, 2004.

[11] SAT benchmarks from Automotive Product Configuration, http://www-sr.informatik.unituebingen.de/~sinz/DC/

[12] C. Sinz, A. Kaiser, and W. Küchlin. “Formal Methods for the Validation of Automotive roduct Configuration Data.” In *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 17 no. 1, pages 75-97, 2003.

[13] L. Zhang and S. Malik. “Extracting small unsatisfiable cores from unsatisfiable Boolean formula.” Presented at the Sixth International Conference on Theory and Applications of Satisfiability Testing, 2003.

Optimizations for Compiling Declarative Models into Boolean Formulas

Darko Marinov[1], Sarfraz Khurshid[2], Suhabe Bugrara[3],
Lintao Zhang[4], and Martin Rinard[3]

[1] Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61801, USA
[2] Dept. of Electrical & Computer Engineering, University of Texas, Austin, TX 78712, USA
[3] MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA 02139, USA
[4] Microsoft Research Silicon Valley Lab, Mountain View, CA 94043, USA
marinov@cs.uiuc.edu
khurshid@ece.utexas.edu
{sbugrara, rinard}@csail.mit.edu
lintaoz@microsoft.com

Abstract. Advances in SAT solver technology have enabled many automated analysis and reasoning tools to reduce their input problem to a SAT problem, and then to use an efficient SAT solver to solve the underlying analysis or reasoning problem. The solving time for SAT solvers can vary substantially for semantically identical SAT problems depending on how the problem is expressed. This property motivates the development of new optimization techniques whose goal is to produce more efficiently solvable SAT problems, thereby improving the overall performance of the analysis or reasoning tool.

This paper presents our experience using several mechanical techniques that enable the Alloy Analyzer to generate optimized SAT formulas from first-order logic formulas. These techniques are inspired by similar techniques from the field of optimizing compilers, suggesting the potential presence of underlying connections between optimization problems from two very different domains. Our experimental results show that our techniques can deliver substantial performance improvement results—in some cases, they reduce the solving time by an order of magnitude.

1 Introduction

In recent years, dramatic advances in the capabilities of SAT solvers have made them an attractive target for model checkers and other automated reasoning systems [13, 3, 14, 24]. The standard approach is to automatically generate a SAT problem, invoke the SAT solver to produce a solution, then transform the SAT solution back into a solution for the initial problem.

The efficiency of this approach depends largely on the efficiency of the SAT solver. However, the SAT solving times can vary by significant factors for semantically identical SAT problems depending on the precise formulation of the SAT problem [7]. This suggests that appropriately optimizing the generated SAT problems may significantly improve the overall performance of systems that use SAT solvers.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 187–202, 2005.

This paper presents our experience with several techniques that are designed to optimize the SAT problems generated by the Alloy Analyzer [10, 12]—a tool that analyzes declarative specifications by translating them into boolean formulas. Our techniques transform a given Alloy specification into an equivalent Alloy specification that induces faster analysis. These transformations are mechanical and therefore suitable for inclusion in an automatic SAT problem generator. Our experimental results show that our techniques can substantially reduce the solving time for a set of benchmark problems from the standard Alloy distribution and previous case studies. The speed-ups range from a low of 1.04X to a high of 14.52X (it is 14.52 times faster to translate and solve the optimized version than the unoptimized version).

Conceptually, our set of transformations draws heavily on techniques from the field of optimizing compilers [2]. We have developed analogs of standard optimizations such as loop unrolling, loop fission and fusion, loop-invariant code motion, common subexpression elimination, constant propagation, and algebraic simplifications. Like their standard compiler optimization counterparts, the goal of these transformations is to reduce the amount of work that the execution engine (the microprocessor or SAT solver) must perform.

We investigated the effect of applying a range of transformations on a suite of benchmark problems. The initial results showed that the most effective transformations were those that focus on formulas that were universally quantified and expressions that use transitive closure. We have implemented these transformations in our prototype tool. The tool allows the user to select a sequence of optimizations which it then applies fully automatically on the given model.

The ostensible purpose of our transformations is to provide an Alloy solver with improved performance. To this end, our transformations focus on specific Alloy constructs (such as transitive closure and quantified formulas) that are responsible for the vast majority of the SAT solving time. (Note the recurring analogy with traditional compiler optimizations, which often focus on loops because programs tend to spend much of their time in loops.) Despite our focus on Alloy constructs, we expect many of the general patterns we exploit in Alloy problems to show up in other domains, which makes our techniques ripe for incorporation into a range of systems that automatically generate SAT problems.

One intriguing aspect of our system is that SAT solvers are a substantially more complex compilation target than the microprocessors that are the traditional compilation targets. In particular, microprocessors often have an available performance model that the compiler writer can use to guide the optimization decisions. The performance of SAT solvers, in contrast, is much less well understood. There are two heuristics in the field: reducing the number of variables tends to reduce the solving time (presumably because it reduces the search space that the SAT solver must explore) and increasing the number of constraints also tends to reduce the solving time (again because it reduces the search space). These are just heuristics so do not hold always [7]. Interestingly enough, one of our performance-improving transformations (constant subexpression elimination) also increases the number of variables. This fact illustrates the need to explore a variety of transformations, not just transformations that are consistent with the current understanding of the performance of SAT solvers.

It is worth emphasizing that the optimizations we propose translate Alloy models into equivalent Alloy models that enable faster analysis—we do not present how to optimize SAT solvers in general. Our approach is inspired by compiler optimizations and can be extended to various other SAT-based techniques, such as bounded model checking [4, 24]. In fact, our approach is not limited to SAT. Any technique that translates a more complicated logic to a simpler logic for reasoning purposes has the potential to benefit from our compiler analogy. As the use of decision procedures becomes more and more popular, we hope this can get more attention from the research community.

This paper makes the following contributions:

- **Optimization Concept:** It introduces the concept of mechanically transforming problems to improve the solving time of the resulting automatically generated SAT problems.
- **Transformations:** It presents a precisely defined set of transformations that optimize the SAT problems that the Alloy Analyzer generates.
- **Implementation:** It presents our implementation, which allows the user to select a sequence of transformations that are then applied fully automatically.
- **Experimental Results:** It presents experimental results that characterize the effectiveness of our optimized SAT problem generator. The results show that our techniques can substantially decrease the solving time for the generated SAT problem.

2 Example

This section illustrates some performance gains that compiler optimizations can provide for SAT-based constraint solving. We present a simple example that models in Alloy a singly-linked acyclic list. We formalize three different constraints that specify acyclicity. We then use the Alloy Analyzer to check that these formulations are equivalent. We re-write the model by applying our optimizations and illustrate how they enable faster analysis. We describe essentials of Alloy as we introduce them. Section 3.1 describes Alloy in more detail.

The following Alloy code declares a list:

```
sig List {
  header: option Entry }

sig Entry {
  next: option Entry }
```

Each list has a `header` entry, and each entry has a `next` entry. The keyword `sig` introduces *signatures*, i.e., basic sets/types. *Fields* in signature declarations introduce relations. The field `header`, for example, introduces the relation `header: List -> Entry`. Further, the keyword `option` constrains this relation to be a partial function.

The following three Alloy *functions* use various Alloy constructs to state the acyclicity constraint:

```
fun Acyclic1(l: List) {
  all e: l.header.*next | e !in e.^next }

fun Acyclic2(l: List) {
  no l.header || (some e: l.header.*next | no e.next) }

fun Acyclic3(l: List) {
  no e: l.header.*next | e -> e in ^next }
```

The *dot operator* (‘.’) represents relational join. Alloy allows intuitive *path expressions* that use transitive closure (‘^’) and reflexive transitive closure (‘*’). For example, `l.header.*next` denotes the set of all entries reachable along the `next` field from the `header` entry of the list `l`.

`Acyclic1` uses universal quantification (‘`all`’), negation (‘`!`’), and set membership (‘`in`’) to state that it is not possible to start a traversal from any list entry and follow one or more fields to get back to the same entry. `Acyclic2` uses existential quantification (‘`some`’) and states that the list is either empty or contains an entry that is reachable from the header and has no next entry. `Acyclic3` uses cross product (‘`->`’) to state that the transitive closure of `next` does not contain any self-loops.

To check equivalence of these definitions, we use the following *assertion*:

```
assert Equiv {
  all l: List {
    Acyclic1(l) => Acyclic2(l)
    Acyclic2(l) => Acyclic3(l)
    Acyclic3(l) => Acyclic1(l) } }
check Equiv for 6
```

The `check` command instructs the Alloy Analyzer to try to generate a counterexample to the given assertion, i.e., a list `l` that does not satisfy the (implicit conjunction of) implication formulas. The analyzer performs the analysis within the given *scope*—a bound on the universe of discourse. In this example, a scope of 6 states that the number of elements in each basic set (known also as “atoms in the signature”) should be at most 6, i.e., at most 6 `Entry` and `Object` elements.

The analyzer takes 199.91 seconds to check `Equiv` and reports that no counterexample exists for this assertion. We next illustrate how compiler optimizations could improve the analyzer’s performance.

Our first optimization is inspired by common subexpression elimination (CSE) [2]. Note that the transitive closure operator (‘^’) appears in the formula body of `Acyclic1` and also of `Acyclic2`. A naive translation of our Alloy assertion to a boolean formula would translate each of these expressions independently. However, they represent the same value and we can apply CSE.

The CSE transformation *adds* a new state component in our model: it introduces a new field, `nextPlus`, in the declaration of `Entry` and constrains it to be the transitive closure of `next`.

```
sig Entry {
  next: option Entry,
  nextPlus: set Entry }

fact { nextPlus = ^next }
```

Each *fact* specifies a constraint that all solutions to the model must satisfy. We next replace each occurrence of `^next` in the functions with `nextPlus`. The analyzer now takes 138.03 seconds to check `Equiv` and (as before) reports no counterexample. It is worth pointing out that by applying an optimization that adds a new state component and is therefore seemingly counter-intuitive (since it increases the size of the underlying state space), we have achieved a reduction in the solving time.

The above optimization factors out the transitive closure operation. We can, in fact, represent the closure directly by its definition; for our example, in the scope of 6, we can *unroll* the closure with respect to this scope and replace the above fact with:

```
fact { nextPlus = next + next.next + next.next.next + next.next.next.next +
                  next.next.next.next.next + next.next.next.next.next.next }
```

where '+' denotes set union in Alloy.

For a given scope, we can also unroll some quantified formulas, besides unrolling the definition of the transitive closure. In particular, we can unroll the formulas, such as `all e: l.header.*next | e !in e.^next`, where the quantified variable ranges over a path expression. (Section 4.1 presents the details of unrolling.) We next apply the loop unrolling transformation, which automatically unrolls all three definitions of acyclicity with respect to scope 6, and check the assertion. The analyzer now takes only 15.84 seconds to check `Equiv` and (as before) reports no counterexample. It is worth pointing out that as a result of this series of optimizations, the Alloy-to-CNF compilation time has gone up from 0.75 seconds to 1.13 seconds; however, in the same time, the SAT-solving time has gone down from 199.16 seconds to just 14.71 seconds.

In summary, even this simple example shows a good potential for optimization: by applying common-subexpression elimination and loop unrolling, we obtain a reduction of more than 92% in the total time to check the formula, i.e., a 12X speedup!

3 Background

This section gives a brief overview of Alloy and SAT technology; following the compiler analogy, Alloy is our input language, and SAT is the target language. We also discuss some of the optimizations that already exist for generating boolean formulas that are likely to induce efficient solving.

3.1 Alloy

Alloy [10] is a first-order declarative language based on sets and relations. The Alloy Analyzer [12] is a tool for automatically analyzing models written in Alloy. The analyzer translates Alloy models into boolean formulas and uses off-the-shelf SAT technology to solve the formulas. Following the compiler analogy, the analyzer consists of the following: a front-end that parses Alloy models into an intermediate representation (IR), a set of optimizations on this IR, and a back-end that translates IR into boolean formulas.

Each Alloy model consists of data (i.e., several sets and relations), several facts (i.e., formulas that put constraints on the data) and an assertion (i.e., a formula to check on the data). These formulas can be structured using functions (i.e., parameterized formulas that can be invoked elsewhere), which the analyzer inlines into the facts and the assertion. Additionally, each analysis specifies a scope (i.e., a bound on the size of basic sets within which to check the formulas). The analyzer translates a conjunction of all facts and the negation of the assertion into a boolean formula such that the boolean formula has a solution iff there are some sets and relations that satisfy all the fact and the negation of the assertion (thus providing a counterexample for the assertion).

Alloy is a relational language; every expression in Alloy denotes a relation (or a set in the case of a relation of arity one). Even the scalars are represented as singleton sets. More details of the Alloy language are available elsewhere [10].

3.2 SAT

Given a propositional formula over a set of boolean variables, the boolean Satisfiability Problem (SAT) asks whether there exists a variable assignment that makes the formula evaluate to true. SAT is a classical NP-Complete problem; therefore, it is unlikely that there is a polynomial algorithm for solving the SAT problem. However, due to its practical importance in areas such as theorem proving, formal verification, and AI planning, much research effort has been put into developing efficient algorithms for solving SAT problems. Although in the worst case these algorithms require exponential time, in practice current state-of-the-art SAT solvers can often determine the satisfiability of boolean formulas with tens of thousands of variables in a reasonable amount of time [27].

Modern SAT solvers determine the satisfiability of a formula by systematically searching the entire boolean space of the formula. They typically require the input formula to be in the Conjunctive Normal Form (CNF), i.e., a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a positive or negative occurrence of a boolean variable. Some recent SAT solvers can operate without the CNF requirement [8], but the Alloy Analyzer translates all formulas into CNF.

Applications that use SAT as the reasoning engine often look like compilers. They take the domain-specific description of the problem as input and translate it into a boolean formula or a sequence of formulas. The formulas are then given to a SAT solver to determine their satisfiability, and the results are fed back to the application to extract meaningful information for the user to understand. For some applications such as circuit verification [17], translation to SAT is straightforward. On the other hand, for applications such as checking the Alloy models, the translation is not trivial [12]. In these applications, different translations of a high-level description may greatly influence the time it takes to solve the output boolean formula.

3.3 Existing Optimizations

There are many heuristics and optimizations developed for making SAT solving efficient in real-world applications. For example, there are several SAT pre-processors [16, 23] that can take a SAT instance and transform it into another form that the SAT solver can easier solve. These transformations happen at the propositional formula level, i.e. after the actual translation has been done. These low-level formulas make it harder or impossible to detect optimizations that can be applied at the higher-level, before the translation. Some researchers tried to influence the output with different translations. For example, Velev [25, 26] tried different encodings of circuit structures for efficient microprocessor verification. Seshia et al. [20] evaluated different encodings for deciding separation logic and proposed a hybrid approach.

Translation has also been studied in the context of Alloy, and the analyzer includes several optimizing translations. Symmetry breaking [21] conjoins new boolean constrains with the input formula to direct the SAT solver's search on non-isomorphic instances. Efficient encoding of (partial) functions [24] replaces the general translation for Alloy relations with a specialized, tighter translation that targets partial functions. Type-based reduction of the number of variables [6] introduces subtyping in Alloy and assigns individual scopes to the subtypes, which partition their supertype, to reduce the overall search space. Narain [19] provides three guidelines for *manual* rewriting of Alloy models to obtain a more efficient analysis but does not consider automation of these rewritings.

A key difference of our work from the previous work is that our optimizations are automatic and at the level of Alloy: they transform an Alloy model to an equivalent Alloy model (without requiring a new type system) that is likely to induce faster solving. This allows our optimizations to be employed in conjunction with existing optimizations in a seamless fashion, thereby increasing the overall performance gain.

4 Optimizations

This section presents some of the optimizations that are applicable to Alloy models. These optimizations are either inspired by or lifted directly from the optimizing compiler literature. We present the following optimizations:

- loop-invariant hoisting (LIH)
- loop fusion (LFU)
- loop unrolling (LUR)
- common subexpression elimination (CSE)
- algebraic transformations (AT)
- partial evaluation (PE)

4.1 Loop Optimizations

Quantifiers (and transitive closure) in Alloy serve a similar purpose as loops in imperative programs. We therefore apply several standard loop optimizations to Alloy formulas that use quantifiers. Each quantified formula introduces a quantified variable (analogous to a loop-index variable) that ranges over some set (analogous to the set of values for the loop index) and has a body formula (analogous to the body of the loop).

Loop-Invariant Hoisting (LIH). This optimization first identifies those subformulas in a body of a given quantified formula that do not depend on the quantified variables and then moves these subformulas out of the quantifier scope. For example, in the Alloy formula `all n: Node | (some Node || n !in n.^next)`, the subformula `some Node` is independent of the variable `n`. Moving the subformula outside of the quantification gives the formula `some Node || all n: Node | n !in n.^next`. This optimization does not give a substantial speedup if the hoisted subformula is much simpler than the rest of the body.

Loop Fusion (LFU). Just as the standard loop fission and fusion optimizations split or merge loops (to make them parallelizable or eliminate the overhead of checking branches), we can split or merge several Alloy quantifiers that range over the same set. For example, the Alloy formulas `(all n: N | F1(n)) && (all n: N | F2(n))` and `(all n: N | (F1(n) && F2(n)))` are semantically equivalent. It turns out, however, that there can be a significant difference in the solving time for these formulas. We have thus implemented loop fusion in the Alloy Analyzer.

Loop Unrolling (LUR). This optimization targets quantified Alloy formulas where the quantified variables range over some path expression. For instance, the example section shows the formula `all e: l.header.*next | e !in e.^next`. We illustrate how such formulas can be completely unrolled for all elements from `l.header.*next`.

Consider the general formula `all` x`:` E `|` F, where E is an expression of some basic type/set α, and F is the quantified formula that depends on the variable x. An operational reading of this formula would first evaluate E to some set $\{a_1, \ldots, a_k\} \subseteq \alpha$ and then check F for each a_i substituted for x, i.e., $F[a_i/x]$. However, Alloy formulas need to be translated into declarative SAT, so the translation does not follow the operational view. Instead, the analyzer must translate this formula under the assumption that E can evaluate to any subset of α, i.e., in the general form $\bigwedge_{a\in\alpha} a \in E \Rightarrow F[a/x]$. The analyzer does not use the general form directly, but applies several optimizations [9].

Consider the special case where the analyzer can statically enumerate the elements $\{a_1, \ldots, a_k\}$ that E evaluates to in some scope. It can then translate `all` x`:` E `|` F into the Alloy formula F`(`a_1`) && ... &&` F`(`a_k`)`. Universally quantified formulas translate into a conjunction; existentially quantified formulas analogously translate into a disjunction: `some` x`:` E `|` F translates into F`(`a_1`) || ... ||` F`(`a_k`)`.

A common pattern in Alloy formulas is to quantify over path expressions. For such expressions, we cannot enumerate the elements in the set, but we can represent them using the equality on reflexive transitive closure, expressed in Alloy for the example `next` relation: `*next = iden[Node] + next + next.next + ... + next`$^{s-1}$, where s is the scope for `Node`. (A similar equality also holds for transitive closure: `^next = next + next.next + ... + next`^{s}.)

The relations in the path expressions are often (partial) functions, e.g., when modeling fields of object-oriented programs [24]. If `next` is a partial function, `n.next` denotes a set with at most one element when `n` denotes a set with at most one element. The Alloy formula `some n` holds iff the set denoted by `n` is not empty. In the case of `n.next`, `some n.next` holds iff `n` denotes a singleton set, which in Alloy represents a scalar. We can therefore translate `all n: l.header.*next | F(n)` into the following conjunction:

```
some l.header => F(l.header)
some l.header.next => F(l.header.next)
...
some l.header.next^{s-1} => F(l.header.next^{s-1})
```

where the guard `some n` ensures that we preserve the semantics of Alloy in this translation. In the general case, `next` could be an arbitrary relation, and the translation would need to use `one n` (instead of `some n`) to specify that `n` has exactly one element. However, the translations of `some n` and `one n` into boolean formulas differ significantly, and the translation of `some n` is much more efficient than `one n`.

Similarly, we can translate `some n: l.header.*next | F(n)` into the disjunction of `(some l.header.next`^{i} `&& F(l.header.next`^{i}`))` for $0 \le i \le s-1$.

To illustrate an application of loop unrolling, let us reconsider the list declaration from Section 2 and checking the equivalence of formulas for acyclicity. We can apply loop-unrolling to rewrite `Acyclic1` as:

```
fun Acyclic1(l: List) { // all n: l.header.*next | n !in n.^next
  some l.header => l.header !in l.header.^next
  some l.header.next => l.header.next !in l.header.next.^next
  some l.header.next.next => l.header.next.next !in l.header.next.next.^next
  some l.header.next.next.next =>
    l.header.next.next.next !in l.header.next.next.next.^next
  some l.header.next.next.next.next =>
    l.header.next.next.next.next !in l.header.next.next.next.next.^next
  some l.header.next.next.next.next.next =>
    l.header.next.next.next.next.next !in l.header.next.next.next.next.next.^next }
```

Our implementation of LUR automatically determines whether it is legal to apply LUR to each loop and applies LUR whenever legal.

4.2 Non-loop Optimizations

We next present the optimizations that do not target quantified Alloy expressions.

Common Subexpression Elimination (CSE). This is a standard compiler optimization that replaces the evaluation of N identical expressions with (1) one evaluation whose result is stored and (2) $N - 1$ reads of the stored result. This optimization effectively trades the space that stores the result for the time to recompute the expressions. Alloy has no operational computation, but it is still desirable to save and reuse results. Others have observed this effect, and the back-end of the Alloy Analyzer actually implements a sophisticated optimization that detects and exploits sharing of subformulas during the translation of quantified Alloy formulas into boolean formulas [22]. However, our results (Section 5) show that a substantial amount of sharing can be detected even in the front-end, at the level of Alloy, before the translation to SAT.[1]

Algebraic Transformations (AT). Alloy expressions offer numerous opportunities for applying algebraic transformations, i.e., using the equational rules of the relational algebra that underlies the Alloy semantics to replace selected expressions with equivalent rewritten expressions. For example, one such rule is that the transitive reflexive closure is idempotent: `**next = *next` for all relations `next`. Our anecdotal experience indicates that Alloy users sometimes write (typically by making a typographic mistake) such expressions that can be significantly optimized. For example, replacing `**next` with `*next` in the formula `**next = iden[Node] + ^next` reduces the checking time by 2X (in scope 7).

Algebraic transformations also apply to the models discussed in Section 5. Moreover, even the transformations that produce expressions of similar complexity, and thus do not look profitable by themselves, can enable the profitable application of other optimizations. For example, one of the algebraic rules is that transpose and reflexive transitive closure commute for example, `*~next = ~*next`. Our implementation applies this rule as a rewrite from left to right to enable CSE to detect more common subexpressions that contain transitive closures.

Partial Evaluation (PE). Partial evaluation is a standard optimization technique for evaluating expressions at compile-time. It generalizes constant folding (which evaluates

[1] We have recently found that we can obtain some of the CSE speed-up by adding to the Alloy model new fields that trigger the existing sharing, without performing the whole CSE.

only basic operations on constant arguments). We illustrate partial evaluation for Alloy models on the following function adapted from [11]:

```
fun modifies(pre, post: State, mods: set Ref) {
  (all r: Ref - mods | F2) &&
  (all r: mods | F1) &&
  F3 }
```

This function is a part of the model that specifies state transitions. Specifically, `modifies` expresses the general constraint that a transition from the `pre` state to the `post` state can modify only references in the set `mods`. Each specific transition instantiates the generic function with the appropriate value for `mods`. Transitions that do not modify any reference thus instantiate `mods` with the empty set: `modifies(s, s', none[Ref])`, where `none[Ref]` represents the empty set.

This constant argument offers the opportunity to partially evaluate `modifies`. We can clone `modifies` and specialize the copy for the empty set: `Ref - mods` evaluates to `Ref`, and the whole second quantification evaluates to `true`:

```
fun modifiesEmpty(pre, post: State) {
  (all r: Ref | F2) &&
  F3 }
```

We then replace the calls involving the empty set with `modifiesEmpty(s, s')`. Standard partial evaluation usually involves cloning of functions. However, the translation of Alloy already inlines all functions[2] so we can instead specialize the inlined formulas.

Although this optimization in theory bears great potential, we found that it is rarely applicable in Alloy models, so we did not implement it. Our experiment with manual application of PE showed a speed-up of 1.16X for the analysis of the model Views [11].

5 Experiments

This section presents the performance results for several Alloy models taken from the Alloy distribution (`http://alloy.mit.edu/`) and from previous case studies that used Alloy models. We evaluate the effectiveness of the optimizations by using our implemented Alloy transformation system to automatically apply the optimizations to these models. We report the total time that the Alloy Analyzer takes to compile and solve the original and optimized models.

We have implemented our optimizations by changing the Java source code of the Alloy Analyzer version 2.0. (The most recent version is 3.0; we have modified 2.0 because 3.0 was unstable when we had started implementing the optimizations.)

We performed all experiments on a Linux machine with a 1.8 GHz Pentium 4 processor using Sun's Java 2 SDK 1.3.1 JVM. We used our modified Alloy Analyzer and the mChaff [18] SAT solver.

[2] Alloy does not support recursive functions, because it needs to generate SAT formulas. Hence, non-termination of inlining or partial evaluation is not an issue.

Table 1. Benchmark models

model	description	solving (#solutions)	optimizations	#ncnb
List	from the example section	satisfiability (0)	CSE, LUR	21
Types	soundness of Java type system	satisfiability (0)	CSE	106
INS	dynamic networks	satisfiability (0)	AT, CSE, LUR	117
ProtonId	patient identity for proton machine	satisfiability (0)	AT, LFU, CSE	164
Life	game of life	satisfiability (1)	AT, LFU, CSE	88
NetConf	network configuration	satisfiability (1)	LUR	192
BinTree	binary search trees	enumeration (1430)	LFU, CSE	62
RedBlack	red-black trees	enumeration (35)	LFU, CSE, LUR	115

5.1 Benchmarks

Table 1 lists the benchmark models that we use to evaluate our optimizations:

- List models singly linked-lists as described in Section 2. It checks the equivalence of the three definitions of acyclicity.
- Types [6] models a part of the Java type system and checks its soundness by checking a subject reduction theorem: statement executions preserve the correspondence between run-time types and the compile-time types.
- INS [14] models the key data structures and algorithms of the Intentional Naming System [1], a resource discovery and service location architecture for mobile networks. It checks partial correctness of these algorithms.
- ProtonId [6] models the tracking of patients in a (proton) radiotherapy facility. It checks that patients are correctly identified.
- Life [22] is a model of Conway's game of life. It simulates multi-step executions of the game on a variable size grid.
- NetConf [19] is a complex Alloy model for network configuration management. It builds network configurations that satisfy given management policies.
- BinTree models simple binary search trees [5, 15].
- RedBlack models red-black trees [5, 15], which implement balanced binary search trees. The last two models enumerate all appropriate trees within a given size.

For each model, we list whether we are just checking satisfiability (i.e., looking for one solution or showing that none exists) or enumerating all possible solutions (which is, for instance, useful in testing [15]). We also list the optimizations that are applied to get from an original model to an optimized version. We finally show the number of non-comment non-blank lines in each original model. This is a simple illustration of the model size and not a measure of the model complexity. (It may be a measure of the complexity of *developing* the model but not *solving* the model.)

5.2 Results

Table 2 summarizes the results that we obtain by applying the optimizations to the different models. For each model, we present corresponding sets of data for both the original (unoptimized) version and the optimized version. We report the number of

Table 2. Comparison of analyses for models with and without optimizations

	unoptimized				optimized				
model	indep. vars	total vars	clauses	time	indep. vars	total vars	clauses	time	speed-up
List	150	3686	19113	199.92	204	4068	20075	15.85	12.61
Types	1888	82713	381816	4.30	2088	77672	357334	3.49	1.23
INS	1295	75725	728022	1208.47	1511	76176	727564	83.21	14.52
ProtonId	1370	37566	237175	3.97	1514	37938	238227	3.83	1.04
Life	351	38312	188423	51.71	927	35844	186088	27.61	1.87
NetConf	770	44848	415018	10.36	770	44700	414574	8.19	1.26
BinTree	146	4504	20064	39.93	274	4056	19024	28.39	1.41
RedBlack	263	7662	34472	136.22	361	7459	34230	27.30	4.99

independent variables in the translated boolean formula, the total number of variables and clauses in the CNF formula, and the time (in seconds) it takes to check (i.e., translate and solve) the formula. We also present the speed-up resulting from the optimizations.

The models for List, Types, INS, and ProtonId all check an assertion; the scopes were set to 6, 12, 6, and 6, respectively. The analyzer does not find any counterexample. For Life and NetConf, the analyzer generates a solution (a simulation of the game and an appropriate network configuration, respectively). We present the time to find the first solution in both the unoptimized and the optimized versions. We used a scope of 12 for Life and a varying scope for different network elements in NetConf (from 4 security tunnels to 6 routers to 12 interfaces). For BinTree and RedBlack, we present the times the analyzer takes to enumerate all corresponding trees with 8 and 7 nodes, respectively. For Types, INS, and Life (all of which are available as models in the Alloy distribution), we ran the analyzer using the maximum scopes for which those models were previously analyzed (as stated in the distribution files). For other models (which are not yet a part of the standard Alloy distribution), we chose sizes that are representative from previous case studies [6, 15, 19]. The performance results indicate the benefits of applying optimizations, both for determining the satisfiability of boolean formulas and for enumerating all solutions.

In all cases the optimizations produce a performance improvement. The speed-up varies from 1.04X to 14.52X. We point out that the optimizations do not necessarily decrease the number of variables (whether independent or total) or increase the number of clauses. In fact, for INS (for example) the optimizations both increase the number of (independent and total) variables and decrease the number of clauses. Nevertheless, the benchmark still exhibits a more than 10X speed-up.

Note that we obtain the maximum speed-up on one of the most complex models (INS). INS models recursive algorithms that were implemented in Java [14], which requires non-trivial fixed-point computations to be expressed in Alloy. Several researchers, including the first two authors of this paper, analyzed this model in various projects [14, 6, 22]. During these projects, the model was optimized (by both manual rewriting and some automatically applied SAT-generation optimizations) to reduce the solving time. Based on our previous experience with this model, it came as a great surprise to us that it was possible to obtain such a substantial speed-up. We hypothesize that more complex models would indeed benefit the most from systematic compiler optimizations, in part because the complexity of the model inhibits the manual application of large-scale transformations such as loop unrolling.

We next discuss how different optimizations contribute to the speed-up. For Types and NetConf, the entire speed-up comes from a single optimization—CSE for Types and LUR for NetConf. For List, the overall speed-up is split between CSE and LUR as 33.6% for CSE and 66.4%, for LUR. For INS, the overall speed-up is split between AT, CSE, and LUR as 2.8%, 98.4%, and -1.2%, respectively. In other words, applying all optimizations (AT, CSE, LUR) to INS does not give the best speed-up; namely, applying just AT and CSE, without LUR, gives 17.34X speed-up over the unoptimized version! This problem is well-known for the compiler optimizations: the "optimizations" are heuristics that improve the code in most cases but can also degrade the code in some cases. Choosing the best set of optimizations to apply (and the best order in which to apply them) is an open research problem in compiler community. We plan to investigate this problem in the context of Alloy.

Our experiments include the change in the compilation time that results from the automatic application of the optimizations. Specifically, the time that we present is the sum of execution times of three parts: (1) the compilation time that includes the time required to parse the Alloy model and apply the optimizations, (2) the translation time required to translate the (optimized) model into a boolean formula, and (3) the SAT solving time. For the original models, the compilation time ranges from 0.39 sec (for List) to 2.55 sec (for RedBlack), and the translation time ranges from 0.35 sec (for List) to 1.52 sec (for Types). With all our optimizations, the compilation time increases by at most 0.24 sec (for List), and the translation time increases by at most 0.37 sec (for NetConf).[3] Some optimizations even decrease the total compilation time as also observed in other compilation projects [22]. Our results indicate that the compilation and translation times in the experiments were negligible in comparison with the SAT solving time, so even a small increase in the compilation and translation times can be easily outweighed by a decrease in the SAT solving time.

5.3 Discussion

The experiments indicate that the most beneficial optimizations are common subexpression elimination and loop unrolling. The most commonly factored out expressions were those that were based on transitive closure. Indeed, loop unrollings also involved formulas that used (quantified expressions with) transitive closure. Alloy specifications often use transitive closure and quantifiers. These constructs tend to be heavily used since they are the source of much of Alloy's expressive power (as compared to first-order logic without transitive closure or relational queries without quantification) [10]. These Alloy constructs are also analogous to loops in code. Solving formulas with these constructs, however, is expensive. Our experience indicates that, in much the same way that the optimizations in a regular compiler focus on the most expensive parts of the

[3] Due to our unfamiliarity with the details of the analyzer, our modified version does not translate the internal representation of the model after optimizing it. Instead, our version parses the model, optimizes it, then pretty-prints it and parses it again. Our times do not include pretty-printing and reparsing, since an actual implementation would not have these two steps.

code (i.e., the inner loops), the Alloy compilation should focus on the most expensive constructs (i.e., quantified formulas and formulas involving transitive closure).

It is worth pointing out that we use the analyzer to check the partial correctness of our optimizations. It is conceivable that our optimizations, because of a compiler bug, could incorrectly create optimized models that are not equivalent to the original models. To check the (partial) correctness, we employ two techniques: (1) we check the equivalence of the original and optimized formulas (using the analyzer) and (2) we enumerate all solutions to the original formula and all solutions to the optimized formulas and compare the number of solutions. Even though the second technique does not, in general, check equivalence of the formulas, it increases our confidence in the correctness of the translation and tends to complete more quickly than the first technique. We therefore use it when the first technique times out. Note that determining the equivalence of boolean formulas, which are our compilation target, is decidable. In regular compilers, on the other hand, determining the equivalence of the unoptimized and optimized code is undecidable in general.

6 Conclusions

In many situations, the performance of an analysis or reasoning tool is the critical factor that determines its utility to the user or the size of the problem that it can meaningfully address. Reduction to SAT is an increasingly popular solution technique. Previous results, as well as ours, show that the overall performance of the system depends not only on the inherent capabilities of the underlying SAT solver, but also on how the problem is expressed: semantically identical SAT problems have widely varying solving times.

We have presented a set of mechanical transformations that, for Alloy model checking problems, can substantially reduce the SAT solving time. As currently formulated, these transformations apply specifically to Alloy models. However, they all have direct analogs in the field of traditional compiler optimizations, and we anticipate that other researchers should be able to apply similar optimizations to their systems. The overall result should be a substantial improvement in the performance of systems that use SAT solvers to solve important analysis and reasoning problems.

Acknowledgements

We would like to thank Ilya Shlyakhter for discussions on Alloy Analyzer and Alexandru Salcianu for comments on a previous version of this paper. This work was funded in part by NSF ITR award #0086154 and in part by NSF ITR—SoD award #0438967.

References

1. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, December 1999.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

3. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36thConference on Design Automation (DAC)*, New Orleans, LA, June 1999.
4. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
5. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
6. Jonathan Edwards, Daniel Jackson, Emina Torlak, and Vincent Yeung. Faster constraint solving with subtypes. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2004.
7. Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic SAT-compilation of planning problems. In *IJCAI-97, Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176, Nagoya, Japan, August 1997.
8. Malay K. Ganai, Lintao Zhang, Pranav Ashar, Aarti Gupta, and Sharad Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proc. 39thConference on Design Automation (DAC)*, pages 747–750, June 2002.
9. Daniel Jackson. Automating first-order relational logic. In *Proc. 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, San Diego, CA, November 2000.
10. Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. `http://sdg.lcs.mit.edu/alloy/book.pdf`.
11. Daniel Jackson and Alan Fekete. Lightweight analysis of object interactions. In *Proc. Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, October 2001.
12. Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.
13. Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. European Conference on Artificial Intelligence (ECAI)*, Vienna, Austria, August 1992.
14. Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, Sep 2000.
15. Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, Santa Margherita Ligure, Italy, May 2003.
16. Ines Lynce and Joao P. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proc. the IEEE International Conference on Tools with Artificial Intelligence (ICTAI'03)*, November 2003.
17. Joao P. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proc. the IEEE/ACM Design, Automation and Testing in Europe (DATE)*, pages 145–149, March 2003.
18. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference (DAC)*, June 2001.
19. Sanjai Narain. Network configuration management via model finding. Internal report, Telcordia Research, Piscataway, NJ, Sep 2004.
20. Sanjit A. Seshia, Shuvendu K. Lahiri, and Randal E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions". In *Proc. 40thConference on Design Automation (DAC)*, pages 425–430, June 2003.

21. Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.
22. Ilya Shlyakhter, Manu Sridharan, Robert Seater, and Daniel Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. In *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, May 2003.
23. Sathiamoorthy Subbarayan and Dhiraj K Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In *Proc. 7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, May 2004.
24. Mandana Vaziri. *Finding Bugs Using a Constraint Solver*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2003.
25. M. N. Velev. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 310–315, January 2004.
26. M. N. Velev. Encoding global unobservability for efficient translation to SAT. In *Proc. 7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 197–204, May 2004.
27. Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *Proc. 8thConference on Automated Deduction (CADE)*, July 2002.

Random Walk with Continuously Smoothed Variable Weights

Steven Prestwich

Cork Constraint Computation Centre,
Department of Computer Science,
University College Cork, Ireland
s.prestwich@cs.ucc.ie

Abstract. Many current local search algorithms for SAT fall into one of two classes. Random walk algorithms such as Walksat/SKC, Novelty+ and HWSAT are very successful but can be trapped for long periods in deep local minima. Clause weighting algorithms such as DLM, GLS, ESG and SAPS are good at escaping local minima but require expensive smoothing phases in which all weights are updated. We show that Walksat performance can be greatly enhanced by weighting variables instead of clauses, giving the best known results on some benchmarks. The new algorithm uses an efficient weight smoothing technique with no smoothing phase.

1 Introduction

Local search algorithms have a long history in combinatorial optimization. In recent years they have been applied to SAT problems, usually by treating them as MAX-SAT and trying to minimise the objective function (the number of clause violations) to zero. Local search for SAT has steadily improved and is an active area of research.

Some local search algorithms fall into the category of *random walks*, which were a significant advance over earlier successful algorithms such as GSAT [26] and those of Gu [10]. The best-known such algorithm is Walksat [17, 25] which has a number of variants. Walksat/G randomly selects a violated clause then *flips* the variable (reassigns it from true to false or vice-versa) that minimizes the total number of violations. Walksat/B selects flips that incur the fewest *breaks* (non-violated clauses that would be violated by the flip). Both select a random variable in the clause (a *random walk move*) with probability p (the *noise parameter*). Walksat/SKC (Selman-Kautz-Cohen) is a version of B that allows *freebies* to override the random walk heuristic. Freebies are flips that incur no breaks, and if at least one freebie is possible from a violated clause then a random one is always selected. HWSAT [8] is a version of G that breaks ties by preferring the least recently flipped variable, based on a similarly modified GSAT called HSAT [7]. Novelty and R-Novelty use similar but more complex criteria. These were later elaborated to the Novelty+ and R-Novelty+ variants [12] that

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 203–215, 2005.

use occasional random walk steps to avoid stagnation, and are among the most competitive local search algorithms. In TABU search [13, 15, 17] variables that were flipped less recently than a threshold number of flips ago (the *tenure*) cannot be flipped. The tenure replaces the noise parameter of random walk algorithms. In the Iterated Robust Tabu Search (IRoTS) algorithm [27] variables that have not been flipped for a given period are automatically flipped to avoid stagnation.

Some problems defeat random walk algorithms, but are solved quite easily by an alternative form of local search based on *clause weighting*. These algorithms modify the objective function during search. They attach a weight to each clause and minimize the sum of the weights of the violations. The weights are varied dynamically, making it unlikely for the search to be trapped in local minima. Clauses that are frequently violated tend to be assigned higher weights. An early SAT algorithm of this form was Breakout [19] which increments violated clause weights at local minima. A similar approach was used in the later WEIGHT algorithm [2], and variants of GSAT increment weights at local minima [24] or at each flip [4], and may allow weights to slowly decay [5]. The Discrete Lagrangian Method (DLM) [34] periodically smooths the weights to reduce the effects of out-of-date local minima, and is based on the Operations Research technique of Lagrangian relaxation. MAX-AGE [28] is a simplified DLM with fewer runtime parameters and new heuristics. Guided Local Search (GLSSAT) [18] is related to DLM but differs in detail, and derives instead from work on Neural Networks. Smooth Descent and Flood (SDF) [23] introduced the Machine Learning technique of multiplicative weights, and was later elaborated to the Exponentiated SubGradient (ESG) method [22]. The Scaling And Probabilistic Smoothing (SAPS) algorithm [14] is related to ESG but uses a more efficient smoothing mechanism. The Pure Additive Weighting Scheme (PAWS) [29] is a version of SAPS that increases weights additively instead of multiplicatively.

Clause weighting algorithms are excellent at guiding local search out of local minima by consulting the *clause violation* history. An interesting question is: can we achieve a similar effect by consulting the *variable flip* history? This might lead to useful new heuristics for random walk algorithms, and would have technical advantages discussed in Section 5. We shall describe new flip heuristics for random walks that emulate clause weighting performance by maintaining variable weights. Section 2 examines a simple additive heuristic for variable weights. Section 3 describes a new smoothing technique for both clause and variable weights that requires no smoothing phase, reducing runtime overheads. Section 4 evaluates an algorithm with smoothed variable weights. Section 5 discusses the relative merits of variable and clause weighting, and future work.

2 Additive Variable Weighting

The usual rationale for clause weighting is that it escapes local minima by learning about features in that region of the search space. But recent evidence indicates that this picture is flawed, and that clause weighting acts instead as a diversification mechanism [32]. This suggests that random walk algorithms should be able to emulate clause weighting performance by diversifying the selection of variables in the flip heuristic. An obvious way to do this is to prefer variables that were not flipped recently, but this is an old idea used in several algorithms. HSAT, HWSAT, Novelty and R-Novelty prefer least-recently flipped variables, the latter incorporating a tabu tenure of 1 flip. Tie-breaking flip heuristics were added to GSAT and Walksat using a first-in-first-out rule [6]. The MAX-AGE clause weighting algorithm prefers variables that were flipped longer ago than a threshold number of flips. TABU forbids recently-flipped variables from being selected, and IRoTS additionally flips variables that have not been flipped recently. Flip heuristics preferring variables that were not recently flipped have been well explored, yet they do not seem to compete with clause weighting heuristics in their ability to escape local minima.

Might variable selection be improved by importing ideas from clause weighting? To explore this idea we shall attach a dynamic weight to each variable and experiment with heuristics for updating these weights. To the best of our knowledge this is a new approach, though variable assignment (literal) weights have been used before. In [16] a literal weighting heuristic was proposed to combine local search with the DPLL backtracking procedure. When a local search algorithm fails to solve a problem, literals that occur most often in violated clauses are assigned higher scores, which can be used to guide DPLL in a proof of unsatisfiability. In [35] a variant of Walksat for SAT (and MAX-SAT) weights literals by analysing local minima during short runs. The aim is to estimate the frequency of each assignment in (optimal) solutions, called *pseudo-backbone frequencies*. These frequencies are used as weights in longer runs to guide initial variable assignments, flip selection and violated clause selection. Some versions of Guided Local Search also weight variable assignments, though not GLSSAT.

As a first experiment we add a new tie-breaking heuristic to Walksat/SKC: select flip variables as usual, but break ties (among non-freebies) by preferring the variable that has been flipped least often in the search so far; break further ties randomly. Thus the weight of a variable is the number of times it has been flipped, and we select variables with minimal weight as long as this does not conflict with the SKC flip heuristic. We shall evaluate a C implementation of the new algorithm, which we call VW1. Other algorithms used for comparison are implemented in the UBCSAT system [30]. All experiments are performed on a Dell 2.4 GHz Pentium 4 with SuSE Linux 9.1 kernel 2.6.

2.1 Experiments on Ternary Chains

The *ternary chain* problem T_k studied in [21, 33] contains clauses

$$v_1 \qquad v_2 \qquad v_1 \wedge v_2 \rightarrow v_3 \qquad \ldots \qquad v_{k-2} \wedge v_{k-1} \rightarrow v_k$$

and has a single solution in which all variables are true. This highly artificial problem can be solved in linear time by unit propagation, as in DPLL backtracking or hybrid local search algorithms such as Saturn [20] and UnitWalk [11], and quite quickly by clause weighting (see below). Its interest lies in the fact that random walks solve T_k in a number of flips that is exponential in k. Chain problems are intended to simulate the chains of variable dependency that occur in some highly-structured, real-world problems, and to guide the design of new local search techniques. One such technique is a preprocessing method for adding a small number of redundant clauses [33], which makes chain and other problems much easier to solve.

Why is this problem so hard for random walk? A random walk algorithm selects a variable from a violated clause $\bar{v}_i \vee \bar{v}_{i+1} \vee v_{i+2}$. In a violated clause all literals are false so $v_i = v_{i+1} = T$ while $v_{i+2} = F$. An unbiased random walk selects a variable randomly from these three, so it is twice as likely to flip a variable from true to false than vice-versa. The problem has a single solution in which all variables take the value true, so it is twice as likely to move away from the solution as toward it. (See [33] for a detailed analysis of this and related problems.)

Figure 1 compares VW1 with five other local search algorithms on ternary chains. Each point is the median number of flips over 1000 runs. Optimal values of the SKC, HWSAT and VW1 random walk parameter p are used (found by trying values 0.1–1.0 in steps of 0.1), similarly for the Novelty+ probability parameter used to choose between variables in a clause, and the SAPS smoothing parameter ρ. Novelty+ sets the random walk parameter p to a default value of 0.01. SAPS performance is reported to be robust under different values of its other parameters, and a reactive version called RSAPS varies only ρ [14]. TABU was used with a fixed tabu tenure of 10.

The graphs show that most algorithms scale exponentially. However, VW1 scales polynomially: on a log-log plot (not shown) its graph is a straight line. SAPS also scales polynomially for k up to approximately 100, after which it scales exponentially. It is unclear why this occurs, but we conjecture that multiplicative weights cause small floating-point errors that erase long-term information.

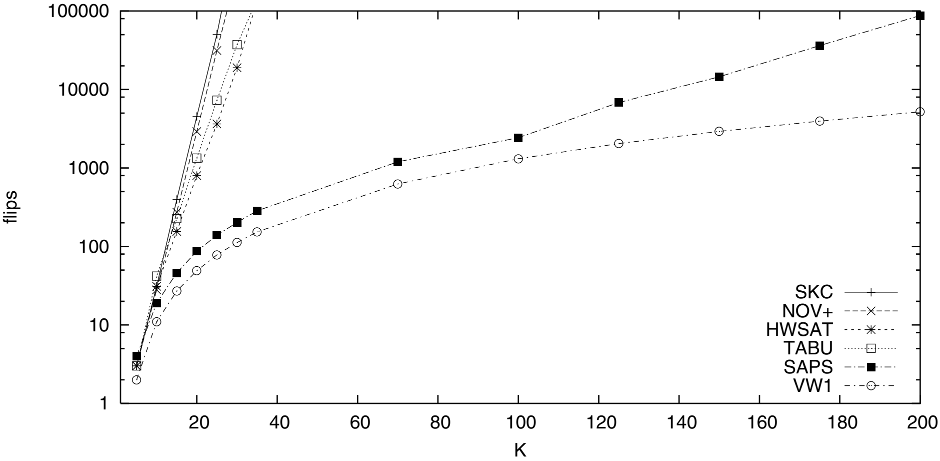


Fig. 1. Local search algorithms on ternary chain problems

HWSAT and TABU use simpler forms of memory and scale only slightly better than Novelty+, which scales slightly better than SKC. Surprisingly, increasing the tabu tenure did not improve TABU. These problems seem to require a particular form of long-term memory currently provided only by clause and variable weighting.

We propose ternary chains as a benchmark for clause and variable weighting algorithms, and other techniques designed to escape local minima. However, it is a very artificial problem so we will also evaluate variable weighting on common SAT benchmarks.

2.2 Experiments on SAT Benchmarks

Next we compare SKC, Novelty+ and SAPS with VW1 on other problems. Figure 2 shows results on selected SAT benchmark problems from the SATLib repository.[1] The figures shown are numbers of flips, taking medians over 100 runs.

instance	SKC	Novelty+	SAPS	VW1
aim50-2.0-1	71933	318514	669	5510
aim50-2.0-2	10727	10338	520	4858
aim50-2.0-3	89040	361853	885	4684
aim50-2.0-4	57098	69040	603	5863
aim100-2.0-1	—	—	4423	324847
aim100-2.0-2	—	—	4898	221535
aim100-2.0-3	—	—	2878	95709
aim100-2.0-4	—	—	5044	175616
aim200-2.0-1	—	—	488547	—
aim200-2.0-2	—	—	181770	3655956
aim200-2.0-3	—	—	298473	—
aim200-2.0-4	—	—	506329	—
logistics.a	64866	46385	6288	36144
logistics.b	88186	54102	5472	25369
logistics.c	104610	81732	7578	37341
logistics.d	425746	117649	33781	183611
bw_large.a	14459	5114	2208	8085
bw_large.b	422723	100830	24649	87843
bw_large.c	8154220	3631196	1904852	582056
bw_large.d	—	5093099	3807160	630464
ais6	891	5388	331	984
ais8	19306	123949	3996	18661
ais10	106752	1523002	18228	101157
ais12	1219293	—	141211	816645
f600	130625	65476	38159	148749
f1000	491262	360709	243377	939022
f2000	2536333	4086895	2849808	—

Fig. 2. Additive variable weighting on SAT benchmarks

[1] http://www.cs.ubc.ca/~hoos/SATLIB/

Again optimal parameter values are used, and in these experiments we tune both the ρ and α SAPS parameters using α values $\{1.1, 1.3, 2.0, 3.0\}$. Figures greater than 10^7 flips are denoted by —.

- The AIM benchmarks [1] are random 3-SAT problems modified to be very hard for random walk. They can be solved with polynomial preprocessing so they are not intrinsically hard. Each instance i is denoted by aim$n$-r-i, meaning that it has n variables and clause/variable ratio r. We use satisfiable instances with clause-variable ratios of 2.0, which are known to be particularly hard for Walksat-style algorithms. Variable weighting greatly boosts SKC performance so that VW1 outperforms SKC and Novelty+, though not SAPS.
- On logistics planning problems VW1 lies between SKC and Novelty+ in performance, and SAPS is again the best algorithm.
- On Blocks World (bw) planning problems VW1 is the best algorithm.
- On All-Interval Series (ais) problems VW1 is similar to SKC, beating Novelty+ but beaten by SAPS.
- On large random 3-SAT problems (f) VW1 is the worst algorithm.

These results show that guiding random walks by variable weighting can greatly boost random walk performance, though not on all problems. Our next step is to import another important technique from clause weighting: smoothing.

3 Phased and Continuous Smoothing

Smoothing techniques for clause weighting algorithms considerably improve performance. Smoothing can be adapted to variable weights, but current clause weighting schemes use expensive *smoothing phases* in which all weights are adjusted to reduce the differences between them. As the number of clauses is often large, this is a significant overhead. We propose a cheaper method with no smoothing phase that can be used for clause or variable weighting.

Smoothing reduces the effect of the earlier search history, placing more emphasis on recent events and adapting the search heuristics to the current search space topology. Our smoothing technique does this as follows. Associate with each variable v a weight w_v, initialised to 0 and updated each time v is flipped according to the formula:

$$w_v \leftarrow (1 - s)(w_v + 1) + s \times t$$

where t denotes time (measured as the number of flips since the start of the search) and a parameter $0 \leq s \leq 1$ controls smoothing. Setting $s = 1$ causes variables to forget their flip history: only a variable's most recent flip has an effect on its weight. Using weights for tie-breaking we obtain an HWSAT-like heuristic. Conversely $s = 0$ causes w_v to behave like a simple counter as in VW1, so that every move in the history has an equal effect. Choosing s between 0 and 1 interpolates between these two extremes, causing older events to have smaller but

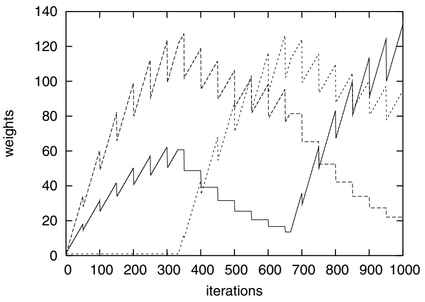


Fig. 3. Phased smoothing

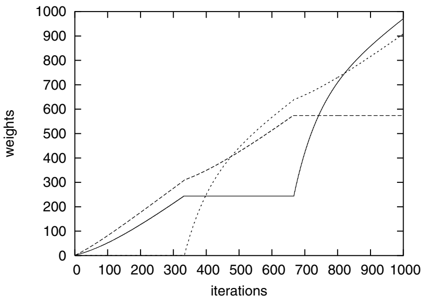


Fig. 4. Continuous smoothing

non-zero effects. We call this *continuous smoothing* because smoothing occurs as weights are updated, without the need for smoothing phases. It also requires only the single s parameter, though when integrating it into a search algorithm in Section 4 we find a further parameter necessary.

To compare phased and continuous smoothing we simulate the evolution of weights during search. In this simulation there are three variables v_1, v_2, v_3 that are flipped during search. The search is divided into three parts. In the first part v_1 is flipped in $\frac{2}{3}$ of the iterations and v_2 in the remaining $\frac{1}{3}$, so variables are flipped in the order $v_1, v_1, v_2, v_1, v_1, v_2 \ldots$ In the second part the same is true of v_2 and v_3 respectively, and in the third part of v_3 and v_1. The simulation shows what happens when variables change from being frequently flipped to rarely flipped, and vice-versa. We use a simple phased weighting method in which weights are increased additively by 1, and smoothed multiplicatively every 50 iterations by 0.8. We compare this with continuous smoothing using $s = 0.02$. The simulations are shown in Figures 3 and 4. The results are qualitatively similar: both methods adjust weight rankings to new situations in almost the same way, though the actual values are different.

Care must be taken when implementing continuous smoothing: if we use integer arithmetic then overflow may occur, though this only occurs on long runs because all weights are bounded above by the number of flips in the search so far. Floating-point arithmetic can be used for long runs, but as weights become very large the term $w_v + 1$ becomes indistinguishable from w_v. A solution is to periodically scale all integer weights down by some factor, similar to a smoothing phase; but whereas phased smoothing typically occurs after a few tens of flips, integer weights (and the flip counter) need not be rescaled more often than every few million flips. Alternatively weights could be implemented using infinite-precision integer arithmetic.

The same method may be applied to clause weighting. At each local minimum we update the weight of each violated clause using the above formula. These weights may then be used as in other clause weighting algorithms. However, in this paper we test only variable weighting.

4 A New Local Search Algorithm

We now describe and evaluate a new local search algorithm called VW2, combining continuously smoothed variable weights with heuristics based on Walksat/SKC. The VW2 algorithm is shown in Figure 5 and is identical to SKC except for its flip heuristic. Instead of using weights for tie-breaking we use them to adjust the break counts as follows. From a random violated clause we select the variable v with minimum score $b_v + b(w_v - M)$ where b_v is the break count of v, w_v is the current weight of v, M is the current mean weight, and c is a new parameter ($c \geq 0$ and usually $c < 1$). Ties are broken randomly. VW2 has three parameters p, s, c but we shall only explore a fraction of the parameter space using p values $\{0.05, 0.1, 0.2, 0.3, 0.4\}$, s values $\{10^{-1}, 10^{-2}, 10^{-3}\}$ and c values $\{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$.

Figure 6 shows results for VW2 on the same benchmarks as in Figure 2 except for the smaller AIM problems, plus two hard graph colouring benchmarks. They are almost uniformly better than those for VW1, and we now compare them to our SKC, Novelty+ and SAPS results in Figure 2; also to published results, where available, for the clause weighting algorithms DLM and MAX-AGE from [28] and SAPS and PAWS from [14, 29] (all using medians over 100 runs).

- On the AIM problems VW2 beats SKC, Novelty+ and SAPS.
- On the logistics planning problems VW2 is beaten by SAPS but beats SKC and Novelty+.
- On the Blocks World planning problems VW2 beats SKC, Novelty+, SAPS, PAWS, DLM and MAX-AGE.
- On the AIS problems VW2 beats SKC and Novelty+, and is comparable to SAPS.
- On the random 3-SAT problems (f) VW2 beats SKC, Novelty+ and (on the largest problem) SAPS, but is beaten by DLM and MAX-AGE.
- On the graph colouring (g) VW2 beats SAPS but is beaten by PAWS, DLM and MAX-AGE.

```
initialise all variables to randomly selected truth values
initialise all variable weights to 0
repeat until no clause is violated
(  randomly select a violated clause C
   if C contains freebie variables
      randomly flip one of them
   else with probability p
      flip a variable in C chosen randomly
   else with probability 1-p
      flip a variable in C chosen by the new heuristic
   update and smooth the weight of the flipped variable
)
```

Fig. 5. The VW2 algorithm

instance	p	s	c	flips	sec
aim200-2.0-1	0.05	0.01	0.001	121735	0.087
aim200-2.0-2	0.05	0.01	0.001	55128	0.040
aim200-2.0-3	0.05	0.01	0.001	66884	0.046
aim200-2.0-4	0.05	0.01	0.001	83576	0.060
logistics.a	0.05	0.1	0.001	13114	0.027
logistics.b	0.05	0.1	0.001	14559	0.034
logistics.c	0.05	0.01	0.001	17734	0.046
logistics.d	0.05	0.01	0.0001	87817	0.21
bw_large.a	0.2	0.01	0.001	5854	0.012
bw_large.b	0.2	0.01	0.0001	73994	0.26
bw_large.c	0.2	0.01	0.00001	508254	3.17
bw_large.d	0.2	0.01	0.000001	570471	5.56
ais6	0.1	0.001	0.1	891	0.0015
ais8	0.1	0.001	0.1	5485	0.013
ais10	0.1	0.001	0.01	27356	0.076
ais12	0.1	0.01	0.001	135394	0.54
f600	0.4	0.1	0.000001	68040	0.10
f1000	0.4	0.1	0.000001	272392	0.53
f2000	0.4	0.1	0.000001	969650	3.05
g125.17	0.2	0.1	0.000001	1820914	29.8
g250.29	0.2	0.1	0.000001	1508571	96.3

Fig. 6. Smoothed variable weighting on SAT benchmarks

The results are clearly competitive with those for current random walk and clause weighting algorithms, and the blocks world planning results are (as far as we know) the best reported. VW2 currently takes an order of magnitude more flips than the best algorithms on 16-bit parity learning problems, but clause weighting algorithms have undergone several generations of development and we hope to emulate their success in future work. We also hope to improve its robustness under different parameter values, following techniques already developed for the Walksat noise parameter.

We now compare flip rates for VW2 and SAPS. SAPS has a more efficient smoothing algorithm than most clause weighting algorithms, but continuous smoothing scales better to large problems. For example on the 600-variable random 3-SAT benchmark the ratio of the VW2 flip rate to that of SAPS is 1.62; on the 1000-variable benchmark it is 1.73; and on the 2000-variable benchmark it is 2.13. Similarly on the Blocks World planning problems the ratio is 1.75 for problem (a), 2.26 for problem (b), 2.80 for problem (c), and 2.99 for problem (d). This is despite the fact that the UBCSAT implementation is generally more efficient than ours. For example on the 600-variable random 3-SAT problem its version of Novelty+ performs roughly 1.42 more flips per second than VW2, on the 1000-variable problem the ratio is 1.78, and on the 2000-variable problem it is 1.94. Similarly on blocks world planning problem (a) the ratio is 1.37, on problem (b) 1.29, on problem (c) 2.10 and on problem (d) 2.95. Combining the implementation techniques of the UBCSAT system with continuous smoothing should yield very scalable algorithms.

5 Discussion

It was recently conjectured that clause weighting works as a form of diversification [32]. We showed that an alternative diversification technique called *variable weighting*, based on variable flip histories, can emulate clause weighting performance. This contributes to the understanding of local search heuristics by supporting the conjecture. It also provides an alternative to clause weighting for problems with deep local minima, and is a promising source of new local search heuristics. We also introduced an efficient new *continuous smoothing* technique for variable weights, which we will test on clause weights in future work.

An advantage of variable weighting is that at each search step only one weight is adjusted, whereas an unbounded (though typically small) number of clause weights may need to be updated at a local minimum. Another advantage is that variable weighting is amenable to *lifting* [9], a technique for compressing a large set of clauses into a single formula via quantification. Lifted clauses are not represented explicitly so dynamically changing weights cannot be assigned to them, but variables can be dynamically weighted. Thus we can achieve clause weighting-like performance on extremely large problems. Variable weighting may also be better suited than clause weighting to weighted MAX-SAT, though this remains to be tested. One of the best local search algorithms on MAX-SAT is currently SAPS [31] and the authors speculate that it will also perform well on weighted MAX-SAT, but note that there are several ways of combining the static clause weights of the problem with the dynamic clause weights of the algorithm. This may be easier with our approach because the static clause weights can be treated separately from the dynamic variable weights.

Finally, a note on completeness. Some local search algorithms have a weak form of completeness called *probabilistic asymptotic completeness* (PAC), meaning that as time tends to infinity the probability of finding a solution tends to one [12]. Does VW2 possess this property? A drawback of freebies is that they make PAC hard to prove, and it is still an open question for SKC except for the special case of 2-SAT [3]. But SKC appears to behave empirically like a PAC algorithm so a proof may eventually be found. Any such proof is likely to reason on freebies and random walk moves alone. VW2 differs from SKC only in its other moves, so we conjecture that VW2 is PAC if and only if SKC is. PAC seems to be less of an issue in clause weighting research, though an inefficient PAC version of Breakout is described in [19].

Acknowledgment

Thanks to the anonymous referees for helpful comments. This material is based in part upon works supported by the Science Foundation Ireland under Grant No. 00/PI.1/C075.

References

1. Y. Asahiro, K. Iwama, E. Miyano. Random Generation of Test Instances with Controlled Attributes. In D. S. Johnson, M. A. Trick (eds), *Cliques, Coloring and Satisfiability: Second Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science* vol. 26, American Mathematical Society 1996, pp. 127–154.
2. B. Cha, K. Iwama. Performance Test of Local Search Algorithms Using New Types of Random CNF Formulas. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann 1995, pp. 304–310.
3. J. Culberson, I. P. Gent, H. H. Hoos. On the Probabilistic Approximate Completeness of WalkSAT for 2-SAT. Technical Report APES-15a-2000, APES Research Group, 2000.
4. J. Frank. Weighting for GODOT: Learning Heuristics for GSAT. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, MIT Press, 1996, pp. 338–343.
5. J. Frank. Learning Short-Term Weights for GSAT. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1997, pp. 384–389.
6. A. S. Fukunaga. Variable-Selection Heuristics in Local Search for SAT. *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, 1997, pp. 275–280.
7. I. P. Gent, T. Walsh. Towards an Understanding of Hill-Climbing Procedures for SAT. *Proceedings of the Eleventh National Conference on Artificial Intelligence*, AAAI Press, 1993, pp. 28–33.
8. I. P. Gent, T. Walsh. Unsatisfied Variables in Local Search. J. Hallam (ed.), *Hybrid Problems, Hybrid Solutions*, IOS Press, Amsterdam, The Netherlands, 1995, pp. 73–85.
9. M. L. Ginsberg, A. J. Parkes. Satisfiability Algorithms and Finite Quantification. *Seventh International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, 2000, pp. 690–701.
10. J. Gu. Efficient Local Search for Very Large-Scale Satisfiability Problems. *Sigart Bulletin* vol. 3, no. 1, 1992, pp. 8–12.
11. E. A. Hirsch, A. Kojevnikov. Solving Boolean Satisfiability Using Local Search Guided by Unit Clause Elimination. *Seventh International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2239, Springer, 2001, pp. 605–609.
12. H. H. Hoos. On the Run-Time Behaviour of Stochastic Local Search Algorithms. *Sixteenth National Conference on Artificial Intelligence*, AAAI Press, 1999, pp. 661–666.
13. W. Huang, D. Zhang, H. Wang. An Algorithm Based on Tabu Search for Satisfiability Problem. *Journal of Computer Science and Technology* vol. 17 no. 3, Editorial Universitaria de Buenos Aires, 2002, pp. 340–346.
14. F. Hutter, D. A. D. Tompkins, H. H. Hoos. Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT. *Eighth International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2470, Springer, 2002, pp. 233–248.
15. B. Mazure, L. Saïs, É. Grégoire. Tabu Search for SAT. *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 1997, pp. 281–285.

16. B. Mazure, L. Saïs, É. Grégoire. Boosting Complete Techniques Thanks to Local Search. *Annals of Mathematics and Artificial Intelligence* vol. 22, 1998, pp. 309–322.
17. D. A. McAllester, B. Selman, H. A. Kautz. Evidence for Invariants in Local Search. *Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, AAAI Press / MIT Press 1997, pp. 321–326.
18. P. Mills, E. P. K. Tsang. Guided Local Search for Solving SAT and Weighted MAX-SAT Problems. *Journal of Automated Reasoning, Special Issue on Satisfiability Problems*, Kluwer, Vol.24, 2000, pp. 205–223.
19. P. Morris. The Breakout Method for Escaping from Local Minima. *Proceedings of the Eleventh National Conference on Artificial Intelligence*, AAAI Press / MIT Press, 1993, pp. 40–45.
20. S. D. Prestwich. Incomplete Dynamic Backtracking for Linear Pseudo-Boolean Problems. *Annals of Operations Research* vol. 130, 2004, pp. 57–73.
21. S. D. Prestwich. SAT Problems With Chains of Dependent Variables. *Discrete Applied Mathematics* vol. 3037, Elsevier, 2002, pp. 1-22.
22. D. Schuurmans, F. Southey, R. C. Holte. The Exponentiated Subgradient Algorithm for Heuristic Boolean Programming. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 2001, pp. 334–341.
23. D. Schuurmans, F. Southey. Local Search Characteristics of Incomplete SAT Procedures. *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, AAAI Press, 2000, pp. 297–302.
24. B. Selman, H. A. Kautz. Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 1993, pp. 290–295.
25. B. Selman, H. A. Kautz, B. Cohen. Noise Strategies for Improving Local Search. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, AAAI Press, 1994, pp. 337–343.
26. B. Selman, H. Levesque, D. Mitchell. A New Method for Solving Hard Satisfiability Problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*, MIT Press, 1992, pp. 440–446.
27. K. Smyth, H. H. Hoos, T. Stützle. Iterated Robust Tabu Search for MAX-SAT. *Proceedings of the Sixteenth Canadian Conference on Artificial Intelligence, Lecture Notes in Computer Science* vol. 2671, Springer Verlag, 2003, pp. 129–144.
28. J. R. Thornton, W. Pullan, J. Terry. Towards Fewer Parameters for Clause Weighting SAT Algorithms. *Proceedings of the Fifteenth Australian Joint Conference on Artificial Intelligence, Lecture Notes in Artificial Intelligence* vol. 2557, Springer-Verlag, 2002, pp. 569–578.
29. J. R. Thornton, D. N. Pham, S. Bain, V. Ferreira Jr. Additive versus Multiplicative Clause Weighting for SAT. *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, San Jose, California, 2004, pp. 191–196.
30. D. A. D. Tompkins, H. H. Hoos. UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT. *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004, pp. 37–46.
31. D. A. D. Tompkins, H. H. Hoos. Scaling and Probabilistic Smoothing: Dynamic Local Search for Unweighted MAX-SAT. *Proceedings of the Sixteenth Canadian Conference on Artificial Intelligence, Lecture Notes in Computer Science* vol. 2671, Springer, 2003, pp. 145–159.

32. D. A. D. Tompkins, H. H. Hoos. Warped Landscapes and Random Acts of SAT Solving. *Proceedings of the Eighth International Symposium on Artificial Intelligence and Mathematics*, 2004 (to appear).
33. W. Wei, B. Selman. Accelerating Random Walks. *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2470, Springer, 2002, pp. 216–232.
34. Z. Wu, B. W. Wah. An Efficient Global-Search Strategy in Discrete Lagrangian Methods for Solving Hard Satisfiability Problems. *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, AAAI Press, 2000, pp. 310–315.
35. W. Zhang, A. Rangan, M. Looks. Backbone Guided Local Search for Maximum Satisfiability. *Eighteenth International Joint Conference on Artificial Intelligence*, 2003, pp. 1179–1186.

Derandomization of PPSZ for Unique-k-SAT

Daniel Rolf

Humboldt-Universität zu Berlin,
Institut für Informatik,
Lehrstuhl für Logik in der Informatik,
Unter den Linden 6, 10099 Berlin, Germany
rolf@informatik.hu-berlin.de

Abstract. The PPSZ Algorithm presented by Paturi, Pudlak, Saks, and Zane in 1998 has the nice feature that the only satisfying solution of a uniquely satisfiable 3-SAT formula can be found in expected running time at most $\mathcal{O}(1.3071^n)$. Using the technique of limited independence, we can derandomize this algorithm yielding $\mathcal{O}(1.3071^n)$ deterministic running time at most.

1 Introduction

The problem of deciding whether a k-CNF G has a satisfying assignment is well known as the k-SAT problem, which is NP-complete for $k > 2$. Hence, if $NP \neq P$ holds (which is widely assumed), there is no hope to find a polynomial time algorithm for the k-SAT problem for $k > 2$.

For a CNF G on n variables, a naive approach is to enumerate all possible assignments and to check for each one whether it satisfies G. This algorithm has $\mathcal{O}\left(poly(|G|) \cdot 2^n\right)$ running time at most. There are significantly more sophisticated algorithms known, and the evolution of expected running time bounds for 3-SAT, which are somewhat below the deterministic ones, is given as [1, 2, 3, 4, 5, 6] with bounds of $\mathcal{O}(1.334^n)$, $\mathcal{O}(1.3302^n)$, $\mathcal{O}(1.32971^n)$, $\mathcal{O}(1.3290^n)$, $\mathcal{O}(1.32793^n)$, and $\mathcal{O}(1.3238^n)$.

In [7], Paturi, Pudlak, Saks, and Zane proved that for a uniquely satisfiable 3-CNF, the solution can be found in $\mathcal{O}(1.3071^n)$ expected running time at most. This is the best randomized bound known for Unique-3-SAT. We refer to their algorithm as the PPSZ Algorithm. But paradoxically, the bound gets worse when the number of solutions increases. This is even more curious since Unique-k-SAT has been proven to be the hardest case of k-SAT for k tending to infinity (cf. [8]). Alas, for the general 3-SAT resp. 4-SAT case, this algorithm achieves expected running time bounds of $\mathcal{O}(1.362^n)$ resp. $\mathcal{O}(1.476^n)$ only, which is worse than the best known randomized bounds of $\mathcal{O}(1.3238^n)$ resp. $\mathcal{O}(1.474^n)$, established in [6] by Iwama and Tamaki.

The best bounds for k-SAT make excessive usage of random bits so that enumerating the entire probability space would yield useless bounds, i.e. much more than $\mathcal{O}(2^n)$. But, do random bounds really compete with deterministic bounds

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 216–225, 2005.

when the existence of true randomness is not provable? At least, randomized algorithms often supply a good starting point to develop fast deterministic algorithms. For example, for k-SAT, the algorithm of Schöning in [1], based on randomized local search and restart, yields a bound of $\mathcal{O}((2 - 2/k + \epsilon)^n)$ expected running time at most, which has been derandomized in [9] to the best known deterministic bound of $\mathcal{O}(1.481^n)$ for $k = 3$ and $\mathcal{O}((2 - 2/(k + 1) + \epsilon)^n)$ for $k > 3$, based on limited local search and covering codes. Alas, like so often, the deterministic bound is much worse than the original randomized one. However, in this paper, we derandomize the PPSZ Algorithm, already mentioned in the paragraph before, for the uniquely satisfiable case yielding (almost) the same bound as the randomized version making it the best known deterministic bound for Unique-k-SAT. We use the technique of limited independence (cf. [10]) to prove that the algorithm can be adapted to enumerate some small probability space yielding a deterministic running time which equals to the former expected running time up to a subexponential factor. Moreover, this means that the best bound for Unique-3-SAT is not only a deterministic one, but also better than the best known randomized bound for (general) 3-SAT.

2 Preliminaries

Firstly, we make some common definitions. A *literal* is a variable or its negation. An assignment β to a set of variables X maps each variable in X to 0 or 1. A literal l is satisfied by β if $X(l) = 1$ if l is not negated resp. $X(\bar{l}) = 0$ if l is negated. A *clause* is a set of literals based on different variables. A clause is satisfied by some assignment β if at least one literal is satisfied by β. A *formula* is a set of clauses. A formula is satisfied by β if each clause is satisfied by β. A *k-clause* is a clause of size k and a *k-CNF* is a set of clauses of size at most k. Finally, a 1-clause is commonly known as *unit clause*. For a set of clauses G, let $vars(G)$ be the set of variables occurring in G.

We will not consider polynomial factors in complexity calculations because we always expect an exponential expression which outweighs all polynomials for large problems, and because the number of clauses is $\mathcal{O}(|vars(G)|^k)$, polynomials that depend on the number of clauses can also be replaced by some polynomial in $|vars(G)|$.

For a CNF G and a literal l, we denote with $G|_l$ the formula obtained by making l true in G, i.e. we remove all clauses that contain l and remove $\bar{l}$ from all clauses that contain it.

A clause pair (C_1, C_2) is a *resolvent pair* if they have only one variable v in common whereby $v \in C_1$ and $\overline{v} \in C_2$. Their *resolvent* $R(C_1, C_2)$ is the clause $(C_1 - v) \cup (C_2 - \overline{v})$. Because any satisfying assignment of C_1 and C_2 must also satisfy $R(C_1, C_2)$, adding $R(C_1, C_2)$ to a CNF does not change its set of satisfying assignments.

s-bounded resolution means to add to G all resolvent pairs of clauses in G where the size of the resolvent is at most s, over and over again until there is

nothing more to do. Note that, if s is a constant, this has polynomial time and space complexity in $|vars(G)|$.

3 The Algorithm

Now, we present our algorithm, which is a derandomized form of the PPSZ Algorithm. Note that π denotes a permutation of the variables of G computed using a polynomial time function $\pi(\alpha)$ where α is a member of some set $\Omega(n, w, L)$. The definition of both objects and the role of the parameters is deferred to Section 4.

Algorithm 1. ***dPPSZ*(*k*-CNF *G*, integer *d*, integer *L*, integer *t*)**

```
$G :=$ do $k^d$-bounded resolution on $G$
for each $\pi = \pi(\alpha)$ with $\alpha \in \Omega(|vars(G)|, (k-1)^{d+1} - 1, L)$ and for each
bit string $b$ of size $t$ do {
    $G' := G$
    repeat as long as there is an unused bit in $b$ {
        $v :=$ next unused variable in $\pi$
        if $G'$ contains a unit clause $v$ resp. $\overline{v}$
        then $G' := G'|_v$ resp. $G' := G'|_{\overline{v}}$
        else Choose $G' := G'|_v$ or $G' := G'|_{\overline{v}}$ depending on the next unused
        bit of $b$ being 1 or 0
    }
    If $G'$ is the empty formula then return $true$
}
return $false$
```

The only difference between the PPSZ Algorithm and this one is that PPSZ choose a permutation π of $vars(G)$ and a bit string b of length $|vars(G)|$ uniformly at random.

4 The Analysis

Without loss of generality, we denote the variables of G with the integers in $[n]$. Moreover, let β denote the one and only satisfying assignment of G.

4.1 Deterministic Bounds for Unique-*k*-SAT

In the algorithm, we use a set $\Omega(n, w, L)$ with $w = (k-1)^{d+1} - 1$, the set will be defined in Section 4.2. But for now, let us use it as a black box probability space that can be used to draw permutations π of $[n]$ at random so that the following lemma is satisfied, which is proved in Section 4.4:

Lemma 2. *Let d and L be integers and let G be a uniquely satisfiable k-CNF G with more than d variables. Fix some variable v of G. Assume that Algorithm 1* *reaches variable v and all variables before v in π were set according to β. At this step, there will be a unit clause for v with probability at least $\lambda_{k,d,L}$ with*

$$\lambda_{k,d,L} = \frac{\mu_k}{k-1} - \epsilon_{k,d,L} \text{ and}$$
$$\mu_k = \sum_{j=1}^{\infty} \frac{1}{j\left(j + \frac{1}{k-1}\right)}$$

where $\epsilon_{k,d,L}$ can be made arbitrary small positive by choosing L and d large enough.

Using linearity of expectation, we can expect to have $\lambda_{k,d,L}n$ unit clauses on the average. Because we try all elements of $\Omega(n, w, L)$, we must encounter at least on permutation π, where the number of unit clauses is at least $\lambda_{k,d,L}n$. Now, assume that the bit string b is chosen so that all bits used for variables agree with β. But, because at least $\lambda_{k,d,L}n$ variables are determined using unit clauses, we only need at most $n - \lambda_{k,d,L}n$ bits from b. So, if we set $t = \lceil n - \lambda_{k,d,L}n \rceil$, we will face that *good* bit string.

Enumerating all bit strings of length t takes time at most $\mathcal{O}(2^t)$. In Section 4.2, we will prove that $\Omega(n, w, L)$ can be constructed and enumerated in polynomial time in $\mathcal{O}(n^{Lw/2})$ which is a polynomial in n for constant k, d, and L. Formulas which do not satisfy the precondition of Lemma 2, i.e. which have at most d variables, can be solved in polynomial time since d is a constant. Finally, we can state:

Proposition 3. *For a uniquely satisfiable k-CNF on n variables, integers $d > 0$, $L > 0$, and $t = \lceil n - \lambda_{k,d,L}n \rceil$, Algorithm 1 finds the satisfying assignment in deterministic running time at most*

$$\mathcal{O}\left(2^{\left(1-\frac{\mu_k}{k-1}\right)n+\epsilon_{k,d,L}n}\right)$$

where $\epsilon_{k,d,L}$ can be made arbitrary small positive by choosing L and d large enough.

Corollary 4. *For a uniquely satisfiable 3-CNF resp. 4-CNF on n variables, the satisfying assignment can be found in deterministic running time at most $\mathcal{O}(1.3071^n)$ resp. $\mathcal{O}(1.4699^n)$.*

4.2 A w-Wise Independent Probability Space for n Reals with Precision L

The (original) PPSZ Algorithm chooses a permutation π uniformly at random, but in Section 4.3, we will see that we need only randomness with respect to a subset of the variables which has size bounded by a constant w for the Unique-k-SAT case. So, we could just draw n integers from a finite pool (to have a finite probability space) of w-wise independent integers and order π according to the rank of these. But, what do we do if we draw the same integer for two variables? Fortunately, the bigger the pool is, the less likely it is for two variables to clash. Guided by this idea, we will discuss a handy construction of π and show some useful properties. For that, our basic tool is Theorem 2.1 from [10–Chapter 15]:

Theorem 5. *For every $n \geq w \geq 1$ there exists a probability space $\Omega(n, w)$ of size $\mathcal{O}(n^{w/2})$ and w-wise independent random variables $y_1, ..., y_n$ over $\Omega(n, w)$ each of which takes* 0 *or* 1 *with probability* 1/2. *$\Omega(n, w)$ can be constructed in polynomial time.*

Let us start with a mapping α which assigns each integer in $[n]$ a real value in $[0, 1)$. We construct a random α in the following way. Define integers $w > 0$, $L > 0$. For each $l \in [L]$ and independently from each other, draw n w-wise independent random variables $y_{1,l}, ..., y_{n,l}$ as stated in the theorem using $\Omega(n, w)$. Define

$$\alpha(v) = \sum_{l \in [L]} 2^{-l} y_{v,l},$$

i.e. $y_{v,.}$ is seen as a binary encoding for $\alpha(v)$ with length L. Let $A^{(L)}$ be the set of all possible real values $\alpha(.)$ can take. For fixed v, the random variables $y_{v,.}$ are fully independent since they are drawn from independent probability spaces. Hence, each value in $A^{(L)}$ has equal probability to be chosen for $\alpha(v)$. On the other hand, for fixed l, the random variables $y_{.,l}$ are w-wise independent since they are drawn using $\Omega(n, w)$. Because this holds for every l independently, the values of α are w-wise independent.

Therefore, the construction above yields w-wise independent α values where each of them takes a value from $A^{(L)}$ uniformly at random. We call this probability space a *w-wise independent probability space for n reals with precision L*, denoted by $\Omega(n, w, L)$. We have:

Lemma 6. *$\Omega(n, w, L)$ can be constructed with size $\mathcal{O}(n^{Lw/2})$ and in polynomial time in its size.*

Given α, we construct a permutation $\pi = \pi(\alpha)$ of $[n]$ so that $\alpha(u) < \alpha(v)$ implies that u occurs before v in π. Such a permutation can clearly be constructed in a deterministic way by ordering $[n]$ due to the values α takes on them with some arbitrary deterministic rule if two take the same value.

Fix some arbitrary $v \in [n]$ and fix some arbitrary $V \subseteq [n] - v$ with $|V| < w$. We want to have a lower bound for the probability that a variable u in V occurs before v in π. The fact that $\alpha(u) = \alpha(v)$ could hold, makes the analysis a little bit complicated. Fortunately, this is not very likely. So, we call v *unique* with respect to V if $\alpha(u) \neq \alpha(v)$ holds for all $u \in V$. Clearly, the probability that v is unique with respect to V is $(1 - 2^{-L})^{|V|}$.

Now, assume that we already know that v is unique with respect to V. All $\alpha(u)$ for $u \in V$ can still be seen as being drawn independently at random from $A^{(L)} - \alpha(v)$. Again, fix a variable u in V. Under the condition that v is unique with respect to V, the probability that $\alpha(u) < \alpha(v)$, i.e. that u occurs before v in π, is equal to $\alpha(v) \cdot 2^L/(2^L - 1)$. This comes from the fact that we have $\alpha(v) \cdot 2^L$ elements in $A^{(L)}$ which are strict less than $\alpha(v)$ and because the condition allows all $2^L - 1$ elements of $A^{(L)} - \alpha(v)$ to be chosen for $\alpha(u)$ uniformly at random. Let us sum up:

Lemma 7. *Let $v \in [n]$ be a variable and $V \subseteq [n]$ be a set of variables with $|V| < w$ and $v \notin V$. Then the following are true:*

1. *The probability that v is unique is $(1 - 2^{-L})^{|V|}$.*
2. *Given that v is unique, all $\alpha(u)$ with $u \in V$ are independent, and for each $u \in V$, the probability that $\alpha(u) < \alpha(v)$ holds is equal to $\alpha(v) \cdot 2^L/(2^L - 1)$.*

4.3 Admissible Trees

Before we can go back to Unique-k-SAT, we need the notion of an admissible tree and have to prove some important properties.

Let T be a tree where the root is labeled by v. Each node of the tree can have a label in $[n]$ or it is unlabeled. Moreover, for each path from a leaf to the root, no integer occurs more than once as a label. Then T is called an *admissible tree.* The depth of T is the maximum distance from any leaf to the root, e.g. a tree containing only one node has depth 0. We limit the depth of an admissible tree to d and we limit the number of children of each node to $k - 1$. Then T has at most $(k-1)^{d+1} - 1$ nodes. A *cut* A is a set of nodes that does not include the root, and every path from the root to a leaf includes a node in A.

Let $\pi = \pi(\alpha)$ where α is drawn from $\Omega(n, (k-1)^{d+1} - 1, L)$ at random. We say a cut A happens if all variables corresponding to labeled nodes of A occur before v in π.

Given that v is unique and a subtree T_0 of T, we denote with $Q_{T_0}(r)$ the probability that at least one of the possible cuts of T_0 happens and conveniently, where we use r to stand for $\alpha(v) \cdot 2^L/(2^L - 1)$. We will establish a lower bound for $Q_{T_0}(r)$:

Lemma 8. *Given an admissible tree T with root labeled by v, let T_0 be some subtree of T with more than one node and let $T_1, ..., T_t$ be the subtrees rooted at the labeled children of the root of T. Let $u_1, ..., u_t$ be the labels of their roots. Then it is true that*

$$Q_{T_0}(r) \geq \prod_{i=1}^{t} (r + (1-r) Q_{T_i}(r))$$

holds where the empty product is interpreted as 1.

Proof. [1] Consider the case that $t = 0$. Since T_0 has at least one child, there is a cut in the tree. But because, no child is labeled, the cut is empty, which

[1] Note that this proof is almost the same like the one for Lemma 4 in [7], which can be found in [11].

corresponds to an empty event, which occurs with probability 1. So, assume $t \geq 1$.

Let U be the set of variables occurring as labels in T. Since $|U| \leq (k-1)^{d+1}-1$ and since we have chosen α from $\Omega(n, (k-1)^{d+1}-1, L)$, we can apply Lemma 7 using $U - v$ for V.

Consider the event that $\alpha(u_i) < \alpha(v)$, which has probability r, and the event that a cut in T_i occurs. Because a subtree of an admissible tree is also admissible, u_i does not occur anywhere else in T_i. Thus, both events are independent, causing their union, denoted with K_i, to have probability $r + (1-r)Q_{T_i}(r)$.

To finish the proof, we have to show that $\mathbb{P}[\bigcap_{i=1}^{t} K_i] \geq \prod_{i=1}^{t} \mathbb{P}[K_i]$. At first, let us recall some standard correlation inequality, which is a special case of the FKG-inequality (cf. Theorem 3.2 in [10–Chapter 6]):

Lemma 9. *Let N be a finite set and let $\mathcal{A}$ and $\mathcal{B}$ be two monotone increasing families of subsets of N, i.e. each super-set of a set in $\mathcal{A}$ resp. $\mathcal{B}$ is also contained in $\mathcal{A}$ resp. $\mathcal{B}$. Draw a random set $M \subseteq N$ by choosing each u in N independently with probability p. Then it is true that*

$$\mathbb{P}[M \in \mathcal{A} \cap \mathcal{B}] \geq \mathbb{P}[M \in \mathcal{A}]\mathbb{P}[M \in \mathcal{B}].$$

We set N to be the set of all variables occurring as labels in T_0, but we exclude v. Moreover, we determine M as follows. For all $u \in N$, we include u in M if it occurs before v in π. These events occur independently each with probability r. Let $\mathcal{W}_i$ denote the family of all subsets of N that imply K_i, i.e. all sets of variables $W \subseteq [n]$ for which holds that K_i happens when all $u \in W$ occur before v in π. Because K_i only depends on variables in N, M is a member of $\mathcal{W}_i$ if and only if the event K_i happens. Clearly, $\mathcal{W}_i$ is monotone increasing since all supersets of a set that implies K_i also imply K_i, i.e. more variables than necessary before v in π is not bad. The set $\mathcal{V}_i = \bigcap_{j=1}^{i-1} \mathcal{W}_i$ is also monotone increasing. We plug $\mathcal{V}_i$ and $\mathcal{W}_i$ as $\mathcal{A}$ and $\mathcal{B}$ in the Lemma and obtain

$$\begin{aligned}\mathbb{P}[M \in \mathcal{V}_{i+1}] &\geq \mathbb{P}[M \in \mathcal{V}_i]\mathbb{P}[M \in \mathcal{W}_i] \\ &\geq \prod_{j \leq i} \mathbb{P}[M \in \mathcal{W}_j].\end{aligned}$$

Because $M \in \mathcal{V}_{t+1}$ means that the event $\bigcap_{i=1}^{t} K_i$ happens, we can conclude that

$$\mathbb{P}\left[\bigcap_{i=1}^{t} K_i\right] \geq \prod_{i=1}^{t} \mathbb{P}[K_i]$$

is true, which completes the proof. □

Let us multiply the probability for the event that v is unique with $Q_T(r)$ to obtain:

Corollary 10. *Given an admissible tree T of depth d with root labeled by v and given that $\alpha(v) = r'$ is true, the probability that a cut of T occurs is at least $Q'_T(r)$ with*

$$Q'_T(r') = Q_T\left(r' \cdot 2^L / \left(2^L - 1\right)\right) \cdot \left(1 - 2^{-L}\right)^{(k-1)^{d+1}-1}.$$

To calculate the probability that at least one cut occurs, we have to average $Q'_T(r')$ with respect to all possible values $r' \in A^{(L)}$. This is given by:

$$2^{-L} \sum_{l=0}^{2^L-1} Q'_T\left(l/2^L\right) = 2^{-L} \sum_{l=0}^{2^L-1} Q_T\left(l/(2^L-1)\right) \cdot \left(1-2^{-L}\right)^{(k-1)^{d+1}-1}$$

Some simple calculations show

$$\lim_{L\to\infty} 2^{-L} \sum_{l=0}^{2^L-1} Q_T\left(l/\left(2^L-1\right)\right) \cdot (1-2^{-L})^{(k-1)^{d+1}-1} = \int_0^1 Q_T(r')dr'.$$

Clearly, the difference between the integral and the sum can be made arbitrary small positive by chosing L large enough. From Section 3.4 in [7], we have that

$$\lim_{d\to\infty} \int_0^1 Q_{T(d)}(r')dr' = \frac{\mu_k}{k-1}$$

where $T(d)$ is an admissible tree of depth d, i.e. we can come arbitrarily close to the right hand side by increasing the depth of T. Combining this with the equation before yields the following result:

Lemma 11. *Given an admissible tree T of depth d, the probability that at least one cut of T occurs is at least $\lambda_{k,d,L}$ with*

$$\lambda_{k,d,L} = \frac{\mu_k}{k-1} - \epsilon_{k,d,L}$$

where $\epsilon_{k,d,L}$ can be made arbitrary small positive by choosing L and d large enough.

4.4 Critical Clause Trees

So, let us draw the connection between Unique-k-SAT and our abstract admissible trees.

We call a clause $C \in G$ a *critical clause* for v if the only true literal in C with respect to β is the one corresponding to v, i.e. flipping the value assigned to v in β would make C instantly false.

Algorithm 1 applies k^d-bounded resolution to G and then steps through the variables ordered by a permutation π. Assume that the bit string b is chosen so that all bits used for variables agree with β. When the algorithm reaches a variable v and there is a critical clause C for v so that the variables $vars(C) - v$ occur before v in π, C has been reduced to a unit clause for v so that the algorithm can immediately determine the right assignment to v. But, when is there a clause C meeting this condition? We need the notion of a critical clause tree.

We call an admissible tree T with root labeled by v a *critical clause tree* for v if for each cut A in T, there exists a critical clause for v in G where $vars(C) - v$ contains only variables which occur as labels in A. The existence of a critical clause tree is given by Lemma 4 in [7]:

Lemma 12. *Let G be a uniquely satisfiable k-CNF with more than d variables. Apply k^d-bounded resolution to G. For each $v \in vars(G)$, there exists a critical clause tree for v with depth d.*

So, if a cut of a critical clause tree of v happens with respect to π, there must be a critical clause C corresponding to that cut meeting the condition. By Lemma 11, this has probability at least $\lambda_{k,d,L}$, and we have proved what was claimed in Lemma 2.

5 Conclusion

We derandomized the uniquely satisfiability case of the PPSZ Algorithm using an approximation of the uniform distribution on $[0,1]$ using a discrete subset of $[0,1]$ and showed that we can come arbitrary close to the randomized bound by making the discrete subset large enough.

We can also conclude that a sufficient pseudo-random number generator can be used for the PPSZ Algorithm instead of true randomness for the unique satisfiability case.

Acknowledgements

The author would like to thank Martin Grohe and the unknown referees for valuable remarks and helpful comments.

References

1. Schöning, U.: A probabilistic algorithm for k-SAT and constraint satisfaction problems. In: Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS). (1999) 410–414
2. Schuler, R., Schöning, U., Watanabe, O.: A probabilistic 3-SAT algorithm further improved. In: Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS). (2002) 192–202
3. Rolf, D.: 3-SAT $\in RTIME(1.32971^n)$. Diploma thesis, Department Of Computer Science, Humboldt University Berlin, Germany (2003)
4. Baumer, S., Schuler, R.: Improving a probabilistic 3-SAT algorithm by dynamic search and independent clause pairs. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT). (2003) 150–161
5. Rolf, D.: 3-SAT $\in RTIME(O(1.32793^n))$ - improving randomized local search by initializing strings of 3-clauses. Electronic Colloquium on Computational Complexity (ECCC) (2003)
6. Iwama, K., Tamaki, S.: Improved upper bounds for 3-SAT. In: Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). (2004) 328–328
7. Paturi, R., Pudlak, P., Saks, M.E., Zane, F.: An improved exponential-time algorithm for k-SAT. In: Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS). (1998) 628–637

8. Calabro, C., Impagliazzo, R., Kabanets, V., Paturi, R.: The complexity of unique k-SAT: An isolation lemma for k-CNFs. In: Proceedings of the 18th Annual IEEE Conference on Computational Complexity (CCC). (2003) 135–141
9. Dantsin, E., Goerdt, A., Hirsch, E.A., Kannan, R., Kleinberg, J., Papadimitriou, C., Raghavan, P., Schöning, U.: A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search. Theoretical Computer Science **289** (2002) 69–83
10. Alon, N., Spencer, J.: The Probabilistic Method. John Wiley (1992)
11. Paturi, R., Pudlak, P., Saks, M.E., Zane, F.: An improved exponential-time algorithm for k-SAT. Journal of the Association for Computing Machinery (JACM) (to appear)
12. Paturi, R., Pudlak, P., Zane, F.: Satisfiability coding lemma. Chicago Journal of Theoretical Computer Science (1999)

Heuristics for Fast Exact Model Counting

Tian Sang, Paul Beame, and Henry Kautz

Computer Science and Engineering,
University of Washington, Seattle WA 98195-2350
{sang, beame, kautz}@cs.washington.edu

Abstract. An important extension of satisfiability testing is model-counting, a task that corresponds to problems such as probabilistic reasoning and computing the permanent of a Boolean matrix. We recently introduced Cachet, an exact model-counting algorithm that combines formula caching, clause learning, and component analysis. This paper reports on experiments with various techniques for improving the performance of Cachet, including component-selection strategies, variable-selection branching heuristics, randomization, backtracking schemes, and cross-component implications. The result of this work is a highly-tuned version of Cachet, the first (and currently, only) system able to exactly determine the marginal probabilities of variables in random 3-SAT formulas with 150+ variables. We use this to discover an interesting property of random formulas that does not seem to have been previously observed.

1 Introduction

The substantial progress in practical algorithms for satisfiability has opened up the possibility of solving many related, and even more difficult, logical problems. In recent work [10], we introduced Cachet, which applies techniques for practical satisfiability algorithms to the associated counting problem, #SAT, that requires the computation of the number of all satisfying assignments of a given CNF formula.

Cachet is an exact model-counting algorithm which combines formula caching [7, 2, 5], clause learning [8, 13, 14], and dynamic component analysis [4, 2, 3]. Cachet was shown to outperform other model counting algorithms on a wide range of problems.

In [10], the primary focus was on managing the component caching, and integrating it with clause learning since the combination of the two can cause subtle errors if certain cross-component implications are not controlled. Handling these problems involved techniques to flush certain cache entries and to detect and prevent cross-component implications involving learned clauses.

In this paper we examine a wide range of different techniques for improving the performance of Cachet, including component-selection strategies, variable-selection branching heuristics, backtracking schemes, and randomization. Many of these techniques have previously worked well in SAT solvers. In addition to studying these heuristics we also study the impact of using a more liberal but still sound method, suggested in [10], controlling cross-components implications involving learned clauses.

One major goal of this work beyond improving the performance of Cachet itself is to determine which of the heuristics from SAT solvers are well-suited for use in #SAT

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 226–240, 2005.

solvers in general. Our results show that some popular techniques such as randomization and aggressive non-chronological backtracking perform quite poorly when combined with component caching as in Cachet and do not appear to be particularly well-suited to use in #SAT solvers in general.

In the case of variable-selection branching heuristics, we observe that in Cachet the tradeoffs between heuristics are somewhat different than in the case of SAT solvers. Based on previous heuristics, we develop a new hybrid branching heuristic, VSADS, which appears to be a good choice for model counting algorithms involving component caching. We also observe that the right application of variable-selection heuristics is secondary to component selection, and we use a new method for selecting components that reduces the amount of wasted effort when the component cache must be flushed. Our experiments also show that the more liberal method for controlling cross-component implications has only a relatively small impact on almost all problems.

Finally, we show how our tuned version of Cachet can be extended to compute all marginal probabilities of variables in random 3-CNF formulas of 150+ variables (at sufficiently high clause-variable ratios). This allows us to discover a new pattern in these marginals. At a clause-variable ratio of roughly 3.4 the conditional probability that a randomly-chosen variable in a satisfying assignment is true is uniformly distributed between 0 and 1. Moreover we derive curves that allow us to predict these probabilities at other ratios. Such results may have explanatory power in the analysis of simple DPLL algorithms.

In the next section we give an overview of Cachet. In Section 3 we discuss the impact of branching heuristics, followed by randomization in Section 4, cross-component implications in Section 5, and non-chronological backtracking in Section 6. Finally, we discuss our methods and results in computing marginal probabilities in Section 7.

2 Overview of Cachet

Cachet, presented in [10], is a practical tool for solving #SAT. It is implemented as a modification and extension of the state-of-the-art SAT solver zChaff [14]. In addition to the 1UIP clause learning of zChaff, Cachet adds two other features that are critical to its performance: an explicit on-the-fly calculation of the connected components of the residual formula at each node in the search tree, and a cache to store the components' model counts so that they do not need to be recalculated when those components reappear later in the search.

Although SAT solvers typically eschew explicit computation of residual formulas, the higher complexity of #SAT complexity means that the sizes of the residual formulas we can deal with are smaller and the benefits of explicit computation outweigh its costs. For the #SAT problems that we can solve using Cachet, the entire overhead of maintaining the residual formulas and calculating connected components is usually roughly half of the total runtime. The learned clauses are used for unit propagations but not considered in the component computation, because their large number would make the component computation much more expensive, and because they would connect subformulas that would otherwise be disjoint components, reducing the advantage of the component decomposition.

Cached components are represented explicitly together with their values in a hash table. The size of this hash table is fixed via an input parameter, and a lazy deletion scheme based on the age of a cached entry is used to keep the table small.

As shown in [10], there can be a very subtle interaction between the component decomposition and unit propagation involving learned clauses. To avoid this, in the original version of Cachet we disallowed any unit propagation inference between two connected components of the residual formula. However, we also showed that this restriction is not strictly necessary and determined the general conditions under which such cross-component unit propagation is safe.

In the original version of Cachet, we also did not use certain features of zChaff, including non-chronological backtracking and its VSIDS variable selection heuristic. Some aspects of non-chronological backtracking as implemented in zChaff are not suitable for model counting. For example, zChaff uses unit propagation from learned clauses rather than explicitly flipping values of decision variables, which works for SAT because all previously explored branches are unsatisfiable; this is not the case for #SAT. We also happened not to use VSIDS because we were exploring heuristics that took advantage of the explicit connected component computation.

In this paper we study the range of options usually considered for SAT solvers and see how they apply in Cachet. These heuristics include branching heuristics as well as randomization and non-chronological backtracking. We also analyze the importance of cross-component implications in component caching context. Finally, we present an extension to Cachet that computes all marginals for satisfiable CNF formulas.

3 Branching

3.1 The Role of Components in Branching Decisions

At any decision-making point, Cachet explicitly maintains the residual formula determined by the current variable assignment, in the form of disjoint components. Thus, at any such point, Cachet can use this partition of variables as part of its branching decisions, information that is not usually available to SAT solvers. Moreover, because these components are disjoint, each component is largely independent of the others. (There is some cross-component information available in the form of learned clauses but, as we will see in section 5, exploiting this information does not have a major impact on the performance of the algorithm.)

Therefore, we separate branching heuristics into two parts: the choice of component and the choice of decision variable/literal within that component. The component selection strategy that the version of Cachet from [10] applied was a pure DFS strategy; that is, only a child of the most recently branched component can be selected as the next component to branch on.

If a component is satisfiable, then all of its child components are satisfiable and it does not matter which child is chosen first; eventually every child component needs to be analyzed, and cached component values are not helpful to their disjoint siblings. However, if a component is unsatisfiable, then at least one of its child components must be unsatisfiable and the values of the others are irrelevant. Naturally, it is preferable to find such a child component first to avoid unnecessary work on satisfiable children. Moreover, not only is the work done on those satisfiable child components useless, but as shown in [10], the calculated values stored in the cache for these components can actually be corrupted by the existence of their not-yet-discovered unsatisfiable sibling and must be explicitly removed from the cache.

Unfortunately, there is no easy predictor for which component will be unsatisfiable. We tried choosing the component with the largest clause/variable ratio, but that was not particularly effective. The solution we have settled on is simple: select the *smallest component first*, measured by number of variables. Because calculating the value of a smaller component is easier, if we do indeed have to abandon this work later because of an unsatisfiable sibling, the amount of wasted effort will be minimized.

We also modified the pure DFS branching order described in [10] so that Cachet attempts to discover unsatisfiable sibling components as quickly as possible. If there are a number of branchable components available, Cachet selects the smallest component C and applies the variable branching heuristics to begin the exploration of C. However, once the first satisfying assignment is found for C, further search in that component is temporarily halted and search within the next smallest remaining component is initiated from that point in the search tree. Once the last child component is found to be satisfiable its analysis is completed and the algorithm backtracks to complete the next-to-last child component, etc. If one of the child components is found to be unsatisfiable, the algorithm backtracks to the point in the search tree where the components were generated. The amount of work in the satisfiable case is still only the sum of the costs of analyzing each component and substantial work may have been saved in the unsatisfiable case.

3.2 Variable Branching Heuristics

Good variable branching heuristics can be critical to the performance of DPLL-based SAT solvers and, since Cachet is a DPLL-based #SAT solver, it is natural that its performance also depends on a good variable branching heuristic. We explore a number of the different branching heuristics available including dynamic literal count heuristics, conflict driven heuristics, and unit-propagation based heuristics. We also develop a new heuristic, VSADS, that seems to be well-suited for #SAT. All these heuristics are currently implemented in Cachet and can be selected by a command line argument. We first review these heuristics.

Literal Count Heuristics. Literal count heuristics [12], make their branching decision based only on the number of occurrences of a variable in the residual formula. If the positive literal $+v$ appears V_p times and the negative literal $-v$ appears V_n times in the residual formula, using a score for variables v as either $V_p + V_n$ or $max(V_p, V_n)$ results, respectively, in the Dynamic Largest Combined Sum (DLCS) and Dynamic Largest Individual Sum (DLIS) heuristics. The highest scored v is selected as the decision variable, and its value is set to true if $V_p > V_n$, false otherwise. The goal is to eliminate as many clauses as possible without considering the impact of unit propagation.

Our original version of Cachet used only these simple heuristics, which are easy to evaluate during component detection. We tried several versions and in our experiments observed that the best was to choose the highest DLCS score with DLIS as a tie-breaker; we refer to this as DLCS-DLIS in our tables of results.

Exact Unit Propagation Count (EUPC) Heuristics. Various unit-propagation-based heuristics have been widely used since early SAT solvers. Such heuristics compute the score of a variable by some magic function over the weights of its positive and negative forms, where a literal's weight is obtained by considering the amount of simplification it yields in unit propagations. Setting proper parameters for such a function is a bit of a black art. In Cachet we tested an EUPC procedure similar to that described for `relsat` [4]. To compute the score of variable v, the EUPC heuristic in ideal form will select a literal whose unit propagation will generate a conflict, and otherwise will choose the best variable score given by the following formula:

$$score(v) = |UP(+v)| \times |UP(-v)| + |UP(+v)| + |UP(-v)|$$

where $UP(\ell)$ is the number of unit propagations induced by setting ℓ to true. Evaluating exact unit propagations for many variables is very expensive, so we also use a preprocessing step as described in [4]. That is, for every variable we compute its approximate score with $|UP(+v)|$ approximated by the number of binary clauses containing literal $-v$ and $|UP(-v)|$ approximated by the number of binary clauses containing literal $+v$. Then the unit propagations and exact scores are computed only for the 10 variables with the best approximate scores.

Approximate Unit Propagation Count (AUPC) Heuristics. By computing a better estimate of the amount of unit propagation that will take place, the AUPC heuristic, suggested in the paper on Berkmin [6], avoids any explicit unit propagations and can be computed more efficiently. The idea is simple: to estimate the impact of assigning $v = 0$ more correctly, not only should the binary clauses containing literal $+v$ be counted, but the binary clauses touching the literals whose negated forms are in binary clauses with v should also be counted. For example, if there is a binary clause $(-u, v)$, when estimating unit propagations resulting from assigning $v = 0$, all the binary clauses containing literal $+u$ should be counted too. The score of a variable is defined as the sum of the scores of its positive form and negative form, and the variable with highest score is chosen as decision variable.

Variable State Independent Decaying Sum (VSIDS). The VSIDS selection heuristic is one of the major successes of Chaff [9, 14]. It takes the history of the search into account but does not analyze the residual formula directly. (This is the reason for the word 'Independent' in its name.) Initially, all variable scores are their literal counts in the original formula. When a conflict is encountered, the scores of all literals in the learned conflict clause are incremented. All variable scores are divided by a constant factor periodically. The idea is to give a higher priority to the literals satisfying recent conflict clauses, which are believed to be more important and necessarily satisfied first. An advantage of VSIDS is its easy score-computing procedure, because it does not require any information from the current residual formula. In fact zChaff does not need to maintain a residual formula.

After many decaying periods, the influence of initial variable scores and old conflicts decay to negligible values and variable scores only depend on recent conflict clauses. If there are very few recent conflicts, then most variables will have very low or even 0 scores, thus decision-making can be quite random. For SAT-solving purposes, this is

Problems	variables	clauses	solutions	DLCS	VSIDS	VSADS	EUPC	AUPC
Circuit								
2bitcomp_6	150	370	9.41E+20	121	43	15	92	112
2bitmax_6	252	766	2.07E+29	189	22	2	21	35
rand1	304	578	1.86E+54	9	X	23	10	16
ra	1236	11416	1.87E+286	3.2	X	3.4	8.2	3.9
rb	1854	11324	5.39E+371	7.1	X	7.5	23	7.9
rc	2472	17942	7.71E+393	172	X	189	736	271
ri	4170	32106	1.30E+719	206	78	119	1296	1571
Grid-Pebbling								
grid-pbl-8	72	121	4.46E+14	0.10	0.10	0.06	0.21	0.15
grid-pbl-9	90	154	6.95E+18	1.1	0.44	0.35	0.35	0.27
grid-pbl-10	110	191	5.94E+23	4.6	0.58	0.37	12	3.2
Logistics								
prob001	939	3785	5.64E+20	0.19	0.13	0.06	0.11	0.06
prob002	1337	24777	3.23E+10	30	10	8	16	21
prob003	1413	29487	2.80E+11	63	15	9	20	35
prob004	2303	20963	2.34E+28	116	68	44	32	133
prob005	2701	29534	7.24E+38	464	11924	331	456	215
prob012	2324	31857	8.29E+36	341	X	304	231	145
flat-200(100)	600	2337						
average	-	-	2.22E+13	102	4.5	4.8	4.6	6.8
median	-	-	4.83E+11	63	2.8	2.9	2.7	3.9
uf200(100)	200	860						
average	-	-	1.57E+9	21	7.1	7.2	3.0	4.8
median	-	-	3074825	18	6.6	6.5	2.5	3.7

Fig. 1. Runtime in seconds of Cachet on a 2.8 GHz Pentium 4 processor with 2 GB memory using various dynamic branching heuristics (X=time out after 12 hours)

not a serious problem, because it probably means the formula is under-constrained and thus easily satisfied. However, in the context of model counting, it is often the case that there are few conflicts in some part of the search tree and in these parts VSIDS will make random decisions.

Variable State Aware Decaying Sum (VSADS). VSADS combines the merits of both VSIDS and DLCS. It is expressly suited for Cachet and can benefit from both conflict-driven learning and dynamic greedy heuristics based on the residual formula. Since all literal counts can be obtained during component detection with little extra overhead, there is no reason for the algorithm not to be "Aware" of this important information for decision-making when most variables have very low VSIDS scores. The VSADS score of a variable is the combined weighted sum of its VSIDS score and its DLCS score:

$$score(VSADS) = p \times score(VSIDS) + q \times score(DLCS)$$

where p and q are some constant factors. Within a component, the variable with the best VSADS score is selected as decision variable. With this derivation, VSADS is expected to be more like VSIDS when there are many conflicts discovered and more like DLCS when there are few conflicts. We report experimental results for $p = 1$ and $q = 0.5$, but the runtime is not particularly sensitive to these precise values.

Experimental Results. Figure 1 shows the results of different heuristics on a number of benchmark problems, including logistics problems produced by Blackbox, satisfiable grid-pebbling problems [10], and benchmarks from SATLIB including circuit problems, flat-200 (graph coloring) and uf200 (3-SAT). The last two sets each contain 100 instances, and the average runtime and the median runtime are given respectively.

Despite being a simple combination of VSIDS and DLCS, VSADS frequently outperforms each of them alone by a large margin. The superiority of VSADS over VSIDS is particularly evident in cases in which VSIDS does not even finish. This is likely because of the random decisions that VSIDS makes when there are few conflicts. In most instances VSADS also significantly improves on DLCS alone. Unit-propagation-count based heuristics EUPC and AUPC are often quite good too, especially on flat-200, uf200 and some logistics problems, but VSADS usually outperforms them and seems to be the most stable one overall. For the remainder of our experiments we report on results using the VSADS heuristic.

4 Randomization

Randomization is commonly used in SAT solvers and appears to be helpful on many problems. In fact, every heuristic discussed before can be randomized easily. The randomization can be either on a tie-breaker among variables with the same score or a random selection of variables whose scores are close to the highest.

We expected that randomization would be somewhat less of a help to #SAT search than for SAT search. One of the major advantages that randomization gives SAT solvers is the ability to find easily searchable portions of the space and avoid getting stuck in hard parts of the space. #SAT solvers, however, cannot avoid searching the entire space and thus might not benefit from this ability. However, it was a little surprising to us that randomization almost always hurts model counting. Figure 2 shows the impact of using randomized VSADS on some problems. VSADS-rand computes variable scores the same way as VSADS does, but it selects a decision variable randomly from the variables whose scores are within 25% of the best score. On the first three problems, we ran VSADS-rand 100 times and used statistical results. Since there are already 100 instances of randomly chosen problems in flat-200 and uf200, we just run VSADS-rand on each problem once. We tried a number of other experiments using other fractions and other heuristics such as EUPC but the results were similar. It is very clear that the effect of randomization is uniformly quite negative. In fact the minimum runtime found in 100 trials is often worse than the deterministic algorithm.

While we do not have a complete explanation for this negative effect, a major reason for this is the impact of randomization on component caching. Using randomization would seem to lower the likelihood of getting repeated components. Consider the search tree on formula F in which there are two large residual formulas A and B that have many variables and clauses in common but are not exactly the same. Since A and B have similar structure, a deterministic branching heuristic is likely to have similar

Problems	grid-pbl-10	prob004	2bitcomp_6	flat-200(100)	uf200(100)
VSADS	0.37	44	15	4.8 / 2.9	7.2 / 6.5
VSADS-rand					
average	6.8	433	54	7.0	12
median	3.7	402	54	3.7	11
maximum	29	1073	81	N/A	N/A
minimum	0.24	145	38	N/A	N/A
STDEV	7.7	200	12	N/A	N/A

Fig. 2. Runtimes in seconds of VSADS versus randomized VSADS on selected problems

variable scores and make similar choices that could easily lead to cache hits involving subproblems of A and B. A randomized heuristic is more likely to create subproblems that diverge from each other and only leads to cache hits on smaller subformulas. Although our experiments showed similar total numbers of cache hits in using randomization, there seemed to be fewer cache hits at high levels in the search tree.

5 Cross-Component Implications

As discussed in our overview, in combining clause learning and component caching we only determine components on the residual formula, not on the the learned clauses. Learned clauses that cross between components can become unit clauses by instantiations of variables within the current component. Unit implications generated in this way are called cross-component implications.

In [10], it was shown that cross-component implications can lead to incorrect values for other components. To guarantee correctness cross-component implications were prohibited; each unit propagation from learned clauses was generated but if the variable was not in the current component the unit propagation was ignored.

However, it was also shown that any implications of literals within the current component that result from further propagations of the literals found in cross-component implications are indeed sound. In prohibiting all cross-component implications, these sound inferences that could help simplify the formula were lost.

In the current version of Cachet such sound implications of cross-component unit propagation are optionally allowed by maintaining a list of cross-component implications. Cross-component implications are detected at the unit-propagation stage, and stored in the list. When branching on a component, it is checked to see if it contains any variable in the list. If the current component has been changed by previous cross-component implications, before branching on it, a new component detection is performed over it, which will update the related data structures correctly. This solution is easy to implement but the overhead can be high, for every element of the cross-component implication list needs to be checked at every decision-making point. Fortunately, the ratio of cross-component implications is very small, at most 0.14% of all implications in our tested formulas, so the overhead is negligible.

Figure 3 shows the impact of cross-component implications. We ran experiments on a much larger suite of problems than are listed; all those not shown have fewer than

Problems	cross-component implications	total implications	time without cross-implications	time with cross-implications
2bitcomp_6	13	3454480	15	15
rand1	6195	5131281	23	23
grid-pbl-8	25	5713	0.06	0.06
prob001	49	16671	0.06	0.06
prob002	225	474770	8	8
prob003	133	426881	9	9
prob004	8988	6575820	50	44
prob005	20607	39391726	482	331
prob012	3674	31862649	313	304
flat-200(100)				
average	277	1010086	4.9	4.8
median	239	767136	2.9	2.9

Fig. 3. Runtime in seconds of VSADS with and without cross-component implications

5 cross-component implications, mostly none. There was one instance in which the speedup using cross-component implications was 46%, but most others were negligible. We conclude that cross-component implication is not an important factor in general.

6 Chronological vs. Non-chronological Backtracking

Non-chronological backtracking is one of the most successful techniques used in SAT solvers. When a clause is learned, the search backtracks to the highest level in the search tree at which the learned clause yields a unit propagation. All decisions made between the level at which the clause is learned and the backtrack level are abandoned since they are in some sense irrelevant to the cause of the conflict found. Since no satisfying assignments have yet been found in that subtree, the only information lost by this abandonment is the path from the backtracking destination to the conflict point, which does not take much time to recover.

However, in the model counting scenario, a direct implementation of the above non-chronological backtracking scheme would abandon work on subtrees of the search tree in which satisfying assignments have already been found and tabulated, and this tabulation would have to be re-computed. This is even worse in the component caching context in which the conflict found may be in a completely separate component from the work being abandoned. Moreover, redoing dynamic component detection and cache checking has an even more significant cost.

As discussed earlier, the basic version of Cachet does have some form of non-chronological backtracking in that finding unsatisfiable components can cause a backtrack that abandons work on sibling components. In contrast to this we use the term far-backtracking to refer to the form of non-chronological backtracking described above.

We considered two forms of far-backtracking, the full original far-backtracking and one in which the backtrack moves up to the highest level below the far backtrack level at which the subtree does not already have a satisfying assignment found. This latter

Problems	# of far backtracks	total #conflicts with far-back	total #conflicts w/o far-back	time with far-back	time w/o far-back
Circuit					
2bitcomp_6	930	1212	1393	15	15
2bitmax_6	1031	1501	1222	6	2
rand1	1270	2114	4095	18	23
ra	0	0	0	3.4	3.4
rb	101	176	220	9	7.5
rc	353	502	511	199	189
ri	13010	13070	10212	164	119
Grid-Pebbling					
grid-pbl-8	75	122	286	0.15	0.06
grid-pbl-9	275	388	773	0.22	0.35
grid-pbl-10	355	506	730	1.8	0.37
Logistics					
prob001	355	506	730	0.07	0.06
prob002	1968	2042	2416	10	8
prob003	2117	2176	2022	14	9
prob004	5657	6694	5492	1062	44
prob005	26233	31392	21951	10121	331
prob012	12020	13563	15677	2860	304
flat-200(100)					
average	6884	6958	7119	4.9	4.8
median	5150	5188	5105	3.0	2.9
uf200(100)					
average	24101	24144	26469	7.5	7.2
median	23014	23067	25778	6.9	6.5

Fig. 4. Runtimes in seconds and number of backtracks using VSADS in Cachet with and without far-backtracking

approach eliminates the problem of abandoned satisfying assignments but it does not backtrack very far and creates additional overhead. It did not make a significant difference so we do not report the numbers for it.

Figure 4 shows the comparison of Cachet with and without far-backtracking. Even with far-backtracking enabled, some of the backtracks are the same as they would be without it, and we report the number of far backtracks separately from the total number of backtracks that are due to conflicts in the case that far-backtracking is turned on. (In SAT algorithms all backtracks are due to conflicts but in model counting most backtracks involve satisfiable left subtrees.)

While far-backtracking occasionally provides a significant improvement in runtime when the input formula is indeed unsatisfiable, overall it typically performed worse than without far-backtracking. As a result, we do not use far-backtracking as the default but we allow it as an option.

7 Computing All Marginals

We now show how our basic exact model counting algorithm Cachet can be modified to compute marginal probabilities for all variables, that is, the fraction of satisfying assignments in which each variable is true. Although Cachet does not maintain explicit information about the satisfying assignments found, we can maintain enough statistics as we analyze each component to determine the overall marginal probabilities. The basic idea requires that each component is associated not only with a weight representing the fraction of all assignments that satisfy it but also with a vector of marginal probabilities for each of its variables.

In Figure 5 we show a simplified recursive algorithm $marginalizeAll$ that returns the count and passes up the marginal probabilities as well as the count. In this simplified version, the left branch is assumed to correspond to the assignment $v = 0$

Algorithm marginalizeAll

```
$marginalizeAll(\Phi, Marginals)$
// returns satisfying probability of formula $\phi$
// marginals of all variables are returned in vector $Marginals$ as well
  if $\Phi$ is empty, return 1
  if $\Phi$ has an empty clause, return 0
  $LeftValue = RightValue = 1/2$                      // initializing
  $initializeVector(LeftMarginals, 0)$
  $initializeVector(RightMarginals, 0)$
  select a variable $v$ in $\Phi$ to branch           // branching
  $extractComponents(\Phi|_{v=0})$
  for each component $\phi$ of $\Phi|_{v=0}$
    $LeftValue \times= marginalizeAll(\phi, LeftMarginals)$
  for each variable $x \in \Phi$
    if $x \in \Phi|_{v=0}$
      $LeftMarginals[x] \times= LeftValue$            // adjusting
    else
      $LeftMarginals[x] = LeftValue/2$
  $LeftMarginals[v] = 0$
  $extractComponents(\Phi|_{v=1})$
  for each component $\phi$ of $\Phi|_{v=1}$
    $RightValue \times= marginalizeAll(\phi, RightMarginals)$
  for each variable $x \in \Phi$
    if $x \in \Phi|_{v=1}$
      $RightMarginals[x] \times= RightValue$          // adjusting
    else
      $RightMarginals[x] = RightValue/2$
  $RightMarginals[v] = RightValue$
  $Marginals = sumVector(LeftMarginals, RightMarginals)$
  $Marginals /= (LeftValue + RightValue)$             // normalizing
  return $LeftValue + RightValue$
```

Fig. 5. Simplified version of algorithm to compute marginal probabilities of all variables

and the parameter $Marginals$ is passed by reference. Variables in different components are disjoint, so their marginals can be calculated separately but finally need to be adjusted by the overall satisfying probability. The marginal of any variable that has disappeared in the simplified formula is just equal to half of the satisfying probability by definition. For the decision variable, only its positive branch should be counted. $LeftValue + RightValue$ is the satisfying probability of Φ and the normalizing factor for all marginals of Φ.

Though described in a recursive fashion, the real implementation of this algorithm works with component caching in the context of non-recursive backtracking that it inherits from zChaff. Moreover, in this simplified version we have ignored the issue of unit propagations. Variables following via unit propagation do not appear in the formula on which a recursive call is made so their marginals are not computed recursively but must be set based on the fraction of satisfying assignments found in the recursive call. The details of this calculation and extension to using arbitrary weights is addressed in [11] where Cachet is extended to handle Bayesian inference.

The overhead of computing all marginals is proportional to the number of variables in the components, rather than the total number of variables, because at a node in the search where the model count is returned, only those relevant variables need to be examined. But it may need a significant amount of memory for caching the marginal vectors. In this way, we are able to compute all marginals quite efficiently, usually with only 10% to 30% extra overhead if the problem fits in the memory.

7.1 Marginals of Random 3-CNF Formulas

In this section we show how the extension of Cachet for computing all marginal probabilities allows us to study new features of random 3-SAT problems. This problem has received a great deal of interest both algorithmically and theoretically. It is known experimentally that there is a sharp satisfiability threshold for random 3-SAT at a ratio of roughly 4.3 clauses per variable. However, the largest proved lower bound on the satisfiability threshold for such formulas is at ratio 3.42 [1] using a very restricted version of DPLL that does not backtrack but makes irrevocable choices as it proceeds. (In fact, almost all analyses of the lower bounds on the satisfiability thresholds for random 3-SAT are based on such restricted DPLL algorithms.)

In Figures 6 and 7 we show the the experimental cumulative distribution of marginals of random 3-CNF formulas of 75 and 150 variables respectively at different ratios. The plots are the result of running experiments with 100 random formulas at each ratio, sorting the variables by their marginals, and taking a subsample of 150 equally-spaced points in this sorted list for ease of plotting the results. Thus the X-axis represents the fraction of all variables considered and the Y-axis represents the marginal probability that a variable is true in satisfying assignments of the formula in which it appears. (Although this is plotted in aggregate, individual formulas have similar plots to these aggregate plots.)

For a cumulative distribution derived in this manner (sometimes called a QQ or quantile-quantile plot), a uniform distribution would be represented as a straight line from $(0, 0)$ to $(1, 1)$. Moreover, we can read off simple properties from these plots.

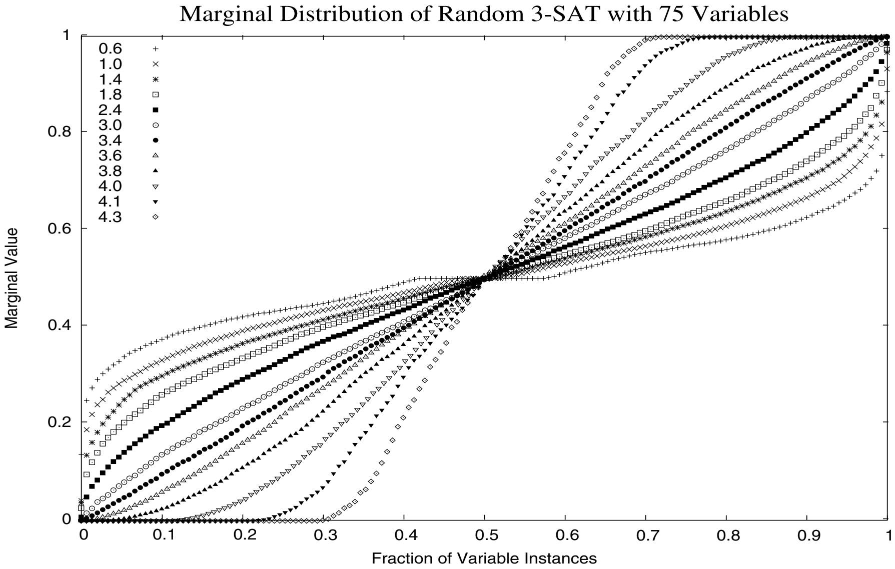


Fig. 6. Cumulative distribution (QQ plot) of marginal probabilities of variables in random 3-CNF formulas of 75 variables at various ratios

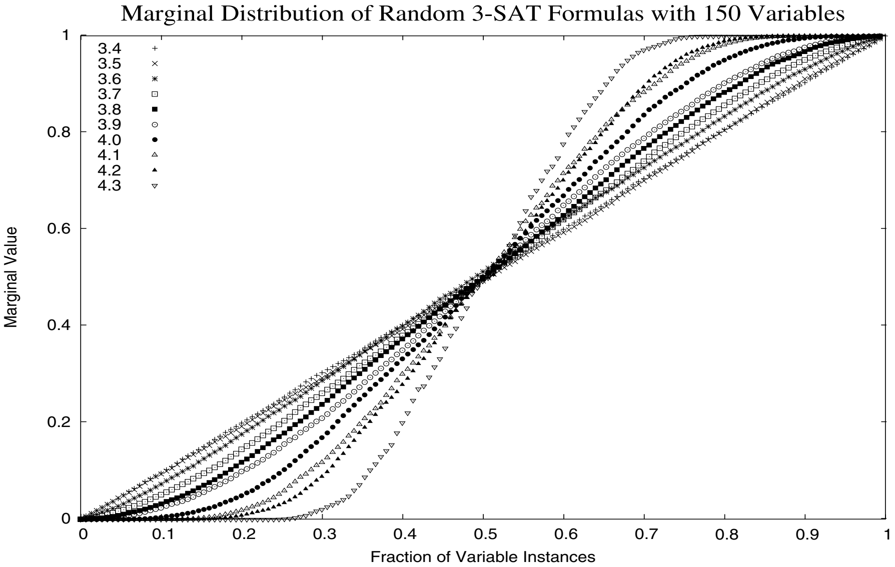


Fig. 7. Cumulative distribution (QQ plot) of marginal probabilities of variables in random 3-CNF formulas of 150 variables at ratios ≥ 3.4

For example, for 75 variables at ratio 4.1, more than 20% of variables were virtually always false in all satisfying assignments and a similar fraction were virtually always true. Since randomly chosen formulas are chosen symmetrically with respect to the signs of their literals, we should expect that the cumulative distribution functions will be symmetric about the point $(0.5, 0.5)$ as is borne out in our experiments.

Our experiments show, not surprisingly, that at low ratios the biases of variables are rarely extreme and that variables become significantly more biased as the ratios increase. At the lowest ratio, 0.6, in Figure 6, a constant fraction of variables do not appear in the formula so the flat section of the curve shows that a constant fraction of variables is completely unbiased at marginal probability 0.5.

In other plots of curves for fixed ratios and varying numbers of variables, we observed that the shape of the curves of the cumulative distribution function seemed to be nearly the same at a given ratio, independent of the number of variables. For example, at ratio 3.9, the lack of smoothness in the plots due to experimental noise almost compensated for any differences in the shapes of the curves.

One interesting property of these cumulative distribution functions is the precise ratio at which the marginal probabilities are uniform. As can be seen from both the 75 variable and 150 variable plots, this point appears to be somewhere around ratio 3.4, although it is a bit difficult to pinpoint precisely. Above this ratio, the marginal probabilities of variables are skewed more towards being biased than unbiased. It seems plausible that the distribution of marginal probabilities is particularly significant for the behavior of non-backtracking DPLL algorithms like the one analyzed in [1]. Is it merely a coincidence that the best ratio at which that algorithm succeeds is very close to the ratio at which the distribution of variable biases becomes skewed towards biased variables?

8 Conclusion

Many of the techniques that apply to SAT solvers have natural counterparts in exact #SAT solvers such as Cachet but their utility in SAT solvers may not be indicative of their utility in #SAT solvers.

We have shown that popular techniques for SAT solvers such as randomization and aggressive non-chronological backtracking are often detrimental to the performance of Cachet. We have developed a new hybrid branching heuristic, VSADS, that in conjunction with a careful component selection scheme seems to be the best overall choice for Cachet. Furthermore, based on experiments, more sophisticated methods for cross-component implications appear to be of only very marginal utility.

Finally, we observed that #SAT solutions are merely the start of what can be obtained easily using Cachet by demonstrating the ability to obtain interesting results on the marginal probabilities of random 3-CNF formulas.

References

1. G. Lalas A. Kaporis, L. Kirousis. The probabilistics analysis of a greedy satisfiability algorithm. In *European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 574–585, 2002.
2. F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and Complexity Results for #SAT and Bayesian inference. In *Proceedings 44th Annual Symposium on Foundations of Computer Science*, Boston, MA, October 2003. IEEE.

3. F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination: Bayesian inference via backtracking search. In *Uncertainty in Artificial Intelligence (UAI-2003)*, pages 20–28, 2003.
4. Roberto J. Bayardo Jr. and Joseph D. Pehoushek. Counting models using connected components. In *Proceedings, AAAI-00: 17th National Conference on Artificial Intelligence*, pages 157–162, 2000.
5. Paul Beame, Russell Impagliazzo, Toniann Pitassi, and Nathan Segerlind. Memoization and DPLL: Formula Caching proof systems. In *Proceedings Eighteenth Annual IEEE Conference on Computational Complexity*, pages 225–236, Aarhus, Denmark, July 2003.
6. E. Goldberg and Y. Novikov. Berkmin: a fast and robust sat-solver. In *Proceedings of the Design and Test in Europe Conference*, pages 142–149, March 2002.
7. S. M. Majercik and M. L. Littman. Using caching to solve larger probabilistic planning problems. In *Proceedings of the 14th AAAI*, pages 954–959, 1998.
8. J. P. Marques Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer Aided Design*, pages 220–227, San Jose, CA, November 1996. ACM/IEEE.
9. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001. ACM/IEEE.
10. Tian Sang, Fahiem Bacchus, Paul Beame, Henry Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.
11. Tian Sang, Paul Beame, and Henry Kautz. Solving bayeian networks by weighted model counting. Submitted, 2005.
12. J. P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, pages 62–74, 1999.
13. Hantao Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction, LNAI*, volume 1249, pages 272–275, July 1997.
14. L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design*, pages 279–285, San Jose, CA, November 2001. ACM/IEEE.

A Scalable Method for Solving Satisfiability of Integer Linear Arithmetic Logic

Hossein M. Sheini and Karem A. Sakallah

Dept. of Electrical Engineering and Computer Science,
1301 Beal Avenue, Ann Arbor, MI 48109-2122, USA
Tel: +1 (734) 996-2528. Fax: +1 (734) 763-4617
{hsheini, karem}@umich.edu

Abstract. In this paper, we present a hybrid method for deciding problems involving integer and Boolean variables which is based on generic SAT solving techniques augmented with a) a polynomial-time ILP solver for the special class of Unit-Two-Variable-Per-Inequality (unit TVPI or UTVPI) constraints and b) an independent solver for general integer linear constraints. In our approach, we present a novel method for encoding linear constraints into the SAT solver through binary "indicator" variables. The hybrid SAT problem is subsequently solved using a SAT search procedure in close collaboration with the UTVPI solver. The UTVPI solver interacts closely with the Boolean SAT solver by passing implications and conflicting assignments. The non-UTVPI constraints are handled separately and participate in the learning scheme of the SAT solver through an innovative method based on the theory of cutting planes. Empirical evidence on software verification benchmarks is presented that demonstrates the advantages of our combined method.

1 Introduction

Many applications in hardware and software verification, such as RTL datapath verification [7], symbolic timing verification [2], and buffer over-run vulnerability detection [32], are naturally cast as decision problems that involve systems of constraints over the Booleans and unbounded integers. In the past several years, there has been considerable progress in solving these types of problems by leveraging the recent advances in Boolean SAT and combining them with methods that decide the feasibility of systems of linear integer constraints.

In this paper we are concerned with quantifier-free *Mixed Integer Boolean* formulas (MIB formulas for short) whose atoms are a) Boolean constants and variables, and b) linear integer constraints. Any such atom is a valid MIB formula, and if φ_1 and φ_2 are valid MIB formulas then so are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, and $\varphi_1 \vee \varphi_2$. We note that a MIB formula reduces to a quantifier-free Presburger (QFP) formula when the set of Boolean atoms is empty. A MIB formula is said to be in *Conjunctive Normal Form* (MIB-CNF) if it is the conjunction of a set of *clauses* each of which is the disjunction of a set of *liter-*

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 241–256, 2005.

als where a literal is either an atom or the negation of an atom. An arbitrary MIB formula can be expressed in MIB-CNF in linear time and space by introducing a set of Boolean auxiliary variables, if necessary [33]. When the distinctions are not important, we will use the terms MIB formula and MIB-CNF formula in the sequel to generically refer to a QFP formula. 2

Current methods for deciding the satisfiability/unsatisfiability of MIB-CNF formulas can be divided into four categories:

1. **SAT-Based Abstraction/refinement methods** [4, 20]**.** In these approaches, the MIB-CNF formula φ is *abstracted* to a pure Boolean CNF formula $\hat{\varphi}$ by replacing each linear integer constraint with an unrestricted Boolean *indicator* variable. A SAT solver is then used to check the satisfiability of $\hat{\varphi}$. If $\hat{\varphi}$ turns out to be unsatisfiable, then so is φ since $\hat{\varphi} \geq \varphi$. On the other hand, if $\hat{\varphi}$ is satisfiable, an ILP solver is called to verify the consistency of that solution. This is done by constructing and solving a system of simultaneous linear integer constraints corresponding to the satisfying assignments found for the indicator variables. If the integer constraints are found to be inconsistent, the abstract formula $\hat{\varphi}$ is *refined* by eliminating its current solution and the SAT solver is re-invoked to find another solution. The procedure terminates if the ILP solver establishes the consistency of one of these SAT solutions or proves that all the (potentially exponential) satisfying assignments to $\hat{\varphi}$ are inconsistent. The combination strategies employed in these classes of solvers are either Shostak-like [29] combination methods or the ones based on work by Nelson-Oppen [24]. For a complete survey of these solvers, the reader is referred to [23].
2. **SAT-Based Finite-Domain Instantiation Methods.** Recently there has been increasing interest in a special class of QFP formulas in which most linear constraints are separation constraints of the form $x - y \leq d$; where x and y are integer variables and d is a constant. In [28], the authors computed an upper bound on integer variables and therefore could set the number of bits in each variable in order to reduce the problem to an equi-satisfiable finite-domain and consequently Boolean problem. They showed that the number of bits per integer variable is linearly proportional to the number of non-separation constraints and logarithmically to the number and size of non-zero coefficients. The separation constraints, on the other hand, are pre-processed using the EIJ method [31] which relies on augmenting the Boolean problem with "transitivity constraints" on the values of indicator variables in order to rule out assignments to those variables that do not have any corresponding solutions in the ILP problem. In the worst case, this process can add an exponential number of transitivity constraints depending on the number of separation constraints.
3. **Automata-Based Methods** [15]**.** The key idea here is to convert the MIB formula to a deterministic finite-state automaton (DFA) such that the language of this automaton corresponds to the set of all solutions of the MIB formula. These methods seem to be efficient for formulas whose integer constraints have large coefficients. However, as observed in [15], the effi-

ciency of this technique declines considerably as the number of variables and/or constraints increases.

4. **Integer Linear Programming Methods.** In these techniques, disjunctions are either removed using Big-M method and the result is checked for satisfiability using a Simplex-based ILP solver [10, 17], or they are enumerated in a Disjunctive Normal Form and then solved using Fourier-Motzkin algorithm [26].

The method we introduce in this paper belongs to the SAT-based abstraction/refinement category. Unlike [4, 20], however, in which the SAT and ILP solvers are loosely integrated, our approach tightly integrates a specialized transitive-closure algorithm with the SAT solver. It achieves its performance by taking advantage of the structure of MIB formulas that arise in verification applications. Specifically, it capitalizes on the fact that the majority of the linear integer constraints involve at most two variables that have unit coefficients, namely UTVPI constraints. Such constraints can be checked quickly using a transitive-closure algorithm which is tightly integrated within the search process of a modern SAT solver. The UTVPI solver can yield implications and generate conflict-induced learned constraints in the SAT solver as soon as such combinations are detected. The non-UTVPI constraints, on the other hand, are handled off-line and may yield conflict-induced UTVPI constraints that help to minimize the number of calls on the non-UTVPI solver. For this purpose, a novel strategy based on theory of cutting-planes is adopted in order to learn from the conflicts detected among the non-UTVPI constraints.

These choices work extremely well in practice since the number of UTVPI constraints is usually much larger than the number of non-UTVPI constraints [11, 28]. Unlike the method of [28, 31] whose efficiency heavily relies on having a low number of separation and non-separation constraints, our method while benefitting from the separation of the UTVPI and non-UTVPI solvers, has no such dependencies on the number of constraints.

The remainder of this paper is organized as follows. In Section 2 we cover some preliminaries. In Section 3 our adopted incremental method in solving UTVPI problems is described. Section 4 covers our approach for encoding and solving the MIB problem. Experimental results are reported in Section 5 and we conclude in Section 6.

2 Preliminaries

An integer linear constraint has the general form

$$\sum_{i=1}^{n} a_i x_i \sim b$$

where $a_i, b, x_i \in \mathbb{Z}$ and $\sim \in \{>, \geq, <, \leq, =, \neq\}$. The special form $a_i x_i + a_j x_j \leq b$ with $a_i, a_j \in \{0, \pm 1\}$ is referred to as a *unit two-variables-per-inequality* (UTVPI) constraint. Single-variable constraints are treated as UTVPI constraints by introducing a dummy variable with a zero coefficient. A constraint

$$\varphi(A_1,A_2,A_3,A_4,u,v,w,x,y,z) =$$

$$[A_1 \vee (u - w \leq 5)] \wedge \tag{ω_1}$$

$$[A_2 \vee (v + w \leq 6)] \wedge \tag{ω_2}$$

$$[A_3 \vee (z = 0)] \wedge \tag{ω_3}$$

$$[A_4 \vee (u + v \geq 12)] \wedge \tag{ω_4}$$

$$[\neg A_3 \vee \neg A_4] \wedge \tag{ω_5}$$

$$[(x = z + 1) \vee (x = z + 3) \vee (x = z + 5) \vee (x = z + 7)] \wedge \tag{ω_6}$$

$$[(y = z + 2) \vee (y = z + 4) \vee (y = z + 6)] \wedge \tag{ω_7}$$

$$[(u + v - 4x - 4y = 0)] \tag{ω_8}$$

Fig. 1. Example MIB-CNF instance

that has more than two variables or that has non-unit coefficients will be referred to as a non-UTVPI constraint. Note that unit two-variable integer equalities and negated-equalities can be transformed to conjunction and disjunction of two UTVPI constraints respectively.

The modern algorithms for propositional SAT are quite well-known and will not be elaborated here; in particular we assume that the reader is familiar with the concepts of Boolean Constraint Propagation (BCP) and its efficient implementation using watched literals, conflict analysis, clause recording, non-chronological backtracking, the VSIDS decision heuristic, restarts, etc. For details on these, readers are referred to [21, 22].

Notationally, we will use $A_1, A_2, \ldots$ (resp. $B_1, B_2, \ldots$) to denote the original (resp. indicator) Boolean variables in a MIB-CNF formula. Integer variables will be denoted by lower case letters such as $u, x, y, \ldots$. We will use the formula in Figure 1 as a running example throughout the paper.

3 Deciding Systems of UTVPI Integer Constraints

Problems consisting of conjunctions of UTVPI constraints can be decided using generic ILP solvers such as CPLEX [17] or XPRESS-MP [10]. However, the full-scale Simplex techniques adopted in these solvers do not take advantage of the simple structure of UTVPI constraints and cannot be efficiently integrated within the backtrack search process of modern SAT solvers.

The solution method we adopt in this paper for deciding systems of UTVPI integer linear constraints is a polynomial-time transitive closure algorithm proposed by Jaffar et al. [19] which, in turn, is an extension of Shostak's method for TVPI *real* constraints [30].

A set of UTVPI constraints is said to be *transitively closed* if for each pair of constraints sharing a variable with opposite signs there exists an inequality constraint between the two remaining variables. For instance, the transitive closure of $\{x - y \leq d, y + z \leq d'\}$ is $\{x - y \leq d, y + z \leq d', x + z \leq d + d'\}$ and we say that $x + z \leq d + d'$ is *implied by* $x - y \leq d$ and $y + z \leq d'$. In the MIB-CNF context, we will be interested in a dynamically-changing set of UTVPI constraints. To keep such a set transitively closed, whenever a new constraint is added to the set, all its implied constraints must be derived and added. When an implied constraint ends up involving a single variable, it may need to be *tightened* in order to maintain the unit coefficient property. Specifically, the constraint implied by $ax + by \leq d$ and $ax - by \leq d'$ (recall that $a,b \in \{0, \pm 1\}$) is $2ax \leq d + d'$ whose tightening yields $ax \leq \left\lfloor (d + d')/2 \right\rfloor$. It is easy to show that the worst-case complexity of maintaining a tightened and transitively-closed set of UTVPI constraints is quadratic in time and space.

To illustrate consider the following set of UTVPI constraints:

$$C = \{y + z \leq 4\ ,\ x - y \leq 5\ ,\ x + y \leq 2\} \tag{1}$$

The transitively-closed and tightened set of constraints derived from (1) is easily shown to be:

$$\begin{aligned} \text{Trans}(C) &= C \cup \{x + z \leq 9\ ,\ 2x \leq 7\} \\ \text{Tighten}(\text{Trans}(C)) &= C \cup \{x + z \leq 9\ ,\ x \leq 3\} \end{aligned} \tag{2}$$

Jaffar et al. [19] showed that a set of UTVPI integer constraints C is satisfiable iff Tighten(Trans(C)) does not contain a constraint of the form $0 \leq d$ where $d < 0$. An example of an unsatisfiable constraint set is:

$$\begin{aligned} C &= \{y - z \leq 1\ ,\ x - y \leq 1\ ,\ z - x \leq -3\} \\ \text{Tighten}(\text{Trans}(C)) &= C \cup \{x - z \leq 2\ ,\ 0 \leq -1\} \end{aligned} \tag{3}$$

Our MIB-CNF SAT solver maintains a database of transitively closed and tightened constraint sets. Specifically, suppose that in the course of searching for a solution the sequence of UTVPI constraint sets C_1, C_2, C_3, ... is generated. As each such set C_i is produced, the corresponding Trans(Tighten(C_i)) set is computed incrementally by adding/removing any implied constraints when a new constraint is added/removed.

4 The MIB-CNF Solver

The architecture of our proposed MIB-CNF solver is shown in Figure 2. It consists of three separate solvers: a modern Boolean SAT solver, an incremental UTVPI integer solver, and a generic ILP (non-UTVPI) solver. The Boolean solver orchestrates the entire process and interacts very closely with the UTVPI solver. Specifically, the Boolean solver views the UTVPI solver as a "super clause" that participates in the implication and conflict analysis steps of the search process. The non-UTVPI solver, on the other hand, is invoked only when the combined Boolean/UTVPI solver returns a satisfying assignment to

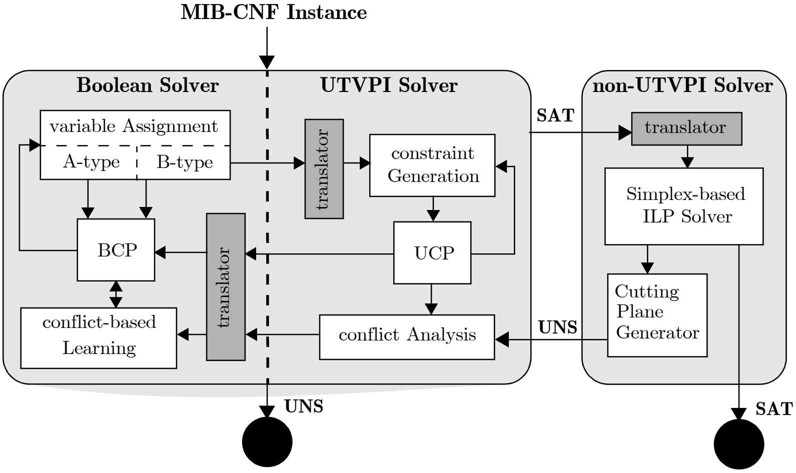


Fig. 2. Overall Architecture of the MIB-CNF Solver

the subset of Boolean and UTVPI constraints. This solver checks the consistency of simultaneous activation of UTVPI and non-UTVPI constraints whose indicator variables are true, by adopting a Simplex-based method. Comparing to other parts of the overall system, the non-UTVPI solver relatively acts as the bottleneck in the process. Therefore, in order to minimize its overall contribution to the run-time of the MIB-CNF solver, two techniques, namely minimizing the size of the ILP problem and learning UTVPI constraints from conflicts in non-UTVPI solver, are adopted. By passing only those absolutely necessary constraints/variables to the non-UTVPI solver, its performance is improved exponentially and by learning from its conflicts in terms of UTVPI constraints, the number of calls to the non-UTVPI solver reduces considerably.

In the remainder of this section we describe how a MIB-CNF instance is encoded for processing by the MIB-CNF solver, and detail the interactions between the Boolean solver and the UTVPI and non-UTVPI solvers.

4.1 Encoding Linear Constraints into the SAT Solver

The first step in solving a MIB-CNF instance is to introduce a set of independent Boolean *indicator* variables B_i to label each of the linear integer constraints and to conjoin the formula with additional relations that establish the equivalence between the indicator variables and their corresponding integer constraints. Specifically, if

$$\varphi = g(A_1,\ldots,A_n,C_1(X),\ldots,C_k(X)) \tag{4}$$

denotes a MIB formula defined over n Boolean variables $A_1, \ldots, A_n$ and k linear integer constraints $C_1(X), \ldots, C_k(X)$, we construct

$$
\begin{array}{llll}
\multicolumn{4}{l}{\hat{\varphi}(A_1, A_2, A_3, A_4, B_1, \cdots, B_8, u, v, w, x, y, z) =} \\
\quad [A_1 \vee B_1] \wedge & (\omega_1) & [B_1 \rightarrow (u - w \leq 5)] \wedge & (\omega_9) \\
\quad [A_2 \vee B_2] \wedge & (\omega_2) & [B_2 \rightarrow (v + w \leq 6)] \wedge & (\omega_{10}) \\
\quad [A_3 \vee B_3] \wedge & (\omega_3) & [B_3 \rightarrow (z = 0)] \wedge & (\omega_{11}) \\
\quad [A_4 \vee B_4] \wedge & (\omega_4) & [B_4 \rightarrow (u + v \geq 12)] \wedge & (\omega_{12}) \\
\quad [\neg A_3 \vee \neg A_4] \wedge & (\omega_5) & [B_{61} \rightarrow (x = z + 1)] \wedge & (\omega_{13}) \\
\quad [B_{61} \vee B_{62} \vee B_{63} \vee B_{64}] \wedge & (\omega_6) & [B_{62} \rightarrow (x = z + 3)] \wedge & (\omega_{14}) \\
\quad [B_{71} \vee B_{72} \vee B_{73}] \wedge & (\omega_7) & [B_{63} \rightarrow (x = z + 5)] \wedge & (\omega_{15}) \\
\quad [1] \wedge & (\omega_8) & [B_{64} \rightarrow (x = z + 7)] \wedge & (\omega_{16}) \\
 & & [B_{71} \rightarrow (y = z + 2)] \wedge & (\omega_{17}) \\
 & & [B_{72} \rightarrow (y = z + 4)] \wedge & (\omega_{18}) \\
 & & [B_{73} \rightarrow (y = z + 6)] \wedge & (\omega_{19}) \\
 & & [1 \rightarrow (u + v - 4x - 4y = 0)] & (\omega_{20})
\end{array}
$$

Fig. 3. Encoding of the MIB-CNF instance of Figure 1. Note that constraint ω_8 does not require an indicator variable since it must be satisfied unconditionally

$$\hat{\varphi} = g(A_1, \ldots, A_n, B_1, \ldots, B_k) \wedge \bigwedge_{i=1}^{k} (B_i = C_i) \tag{5}$$

Clearly, $\varphi = \exists(B_1, \cdots, B_k) \cdot \hat{\varphi}$, and the satisfiability/unsatisfiability of $\hat{\varphi}$ implies that of φ. Strict equivalence between the indicator variables and their corresponding linear integer constraints is not necessary, however. The satisfiability/unsatisfiability of φ can be determined by checking the simpler formula

$$\tilde{\varphi} = g(A_1, \ldots, A_n, B_1, \ldots, B_k) \wedge \bigwedge_{i=1}^{k} (B_i \rightarrow C_i) \tag{6}$$

Since $\tilde{\varphi}$ is a superset of $\hat{\varphi}$, this can be viewed as a conservative abstraction: when $\tilde{\varphi}$ is unsatisfiable, so is $\hat{\varphi}$, and when $\tilde{\varphi}$ is satisfiable, it is possible that $\hat{\varphi}$ is unsatisfiable. However, because of the form of $\hat{\varphi}$ and the fact that, by construction, g is positive unate in all indicator variables B_i, the only situation in which this happens is when at least one B_i is 0 and its corresponding linear integer constraint is forced to be satisfied. Such a solution can be changed so that $B_i = 1$, restoring consistency between the integer constraint and its indicator variable without affecting the satisfiability of the original formula.

The practical effect of checking $\tilde{\varphi}$ rather than $\hat{\varphi}$ is that the linear integer solvers need only process those constraints that are "active," i.e. those whose indicator variables are true; constraints whose indicator variables are false can be safely disregarded. This optimization has a major impact on the performance of the MIB-CNF solver.

Figure 3 demonstrates the result of applying this method of encoding on problem of Figure 1.

4.2 The Boolean/UTVPI Solver Interface

The Boolean and UTVPI solvers interact dynamically during the search process and communicate through the indicator variables of the UTVPI constraints (the non-UTVPI constraints are ignored in this phase.) The Boolean solver is responsible for making decisions on the Boolean variables and for performing BCP and conflict analysis. A decision or implication that sets an indicator variable to 1 triggers the UTVPI solver which, in turn, activates the corresponding linear integer constraint and incrementally updates the database of active UTVPI constraints to keep it transitively-closed and tightened. This update can be viewed as continuing the BCP process within the UTVPI solver, and can yield further implications to other indicator variables or can result in conflicts. Inconsistency of the active UTVPI constraints is communicated back to the Boolean solver as a conflicting assignment on the relevant indicator variables. The Boolean conflict analysis procedure takes over at that point to create an appropriate conflict-induced clause [21] and backtracks to eliminate the conflict. The backtrack level is passed to the UTVPI solver so that it can deactivate the linear integer constraints whose indicator variables were reset to 0 or unassigned (as well as any constraints they implied by transitive closure and tightening).

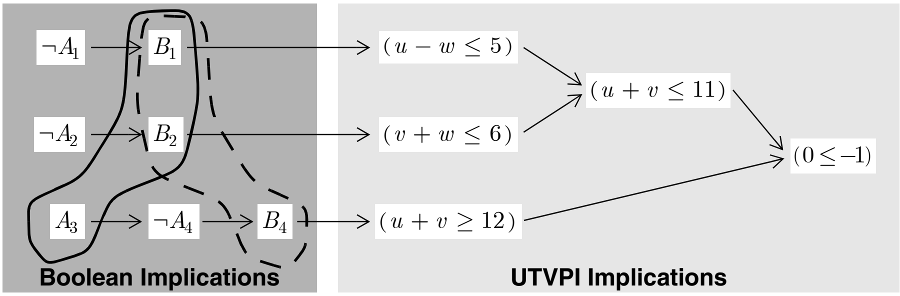


Fig. 4. Implication sequence leading to a conflict in the UTVPI solver. The indicator variables enclosed by the dashed outline are returned by the UTVPI solver as a conflicting assignment. The conflict analysis procedure of the Boolean solver traces back from this assignment to learn and record the conflict-induced clause$(\neg A_3 \vee \neg B_1 \vee \neg B_2 \vee)$

To illustrate the process of implying indicator variables in the UTVPI solver, let B be an indicator variable for a UTVPI constraint C. B is implied to 1 in the UTVPI solver if the process of transitive closure and tightening produces a constraint that is identical or stricter than C. On the other hand, B is implied to 0 if transitive closure and tightening produces a constraint that is inconsistent with C. For example, if B is the indicator variable for $x - y \leq 5$, generating $x - y \leq 3$ causes B to be implied to 1, whereas generating $y - x \leq -6$ causes the implication of B to 0.

The handling of UTVPI conflicts is illustrated in Figure 4 for our running example.

4.3 The Non-UTVPI Solver Theorem 1. Proof:

When the combined Boolean/UTVPI solver returns a satisfying solution, the non-UTVPI solver must be invoked to check the consistency of that solution against any activated non-UTVPI constraints. This can be naively done by collecting *all* of the active linear integer constraints, i.e., the UTVPI and non-UTVPI constraints whose indicator variables are set to 1 in the current solution, and passing them on to a generic Simplex-based ILP solver [10, 17]. As the next theorem shows, however, only a subset of the active UTVPI constraints needs to be processed by the ILP solver.

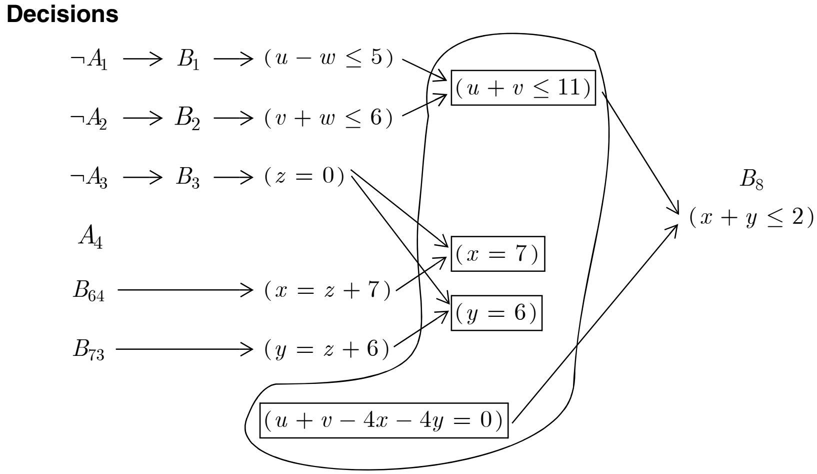


Fig. 5. Conflict analysis in the non-UTVPI solver. After determining that the highlighted constraints are inconsistent, the non-UTVPI solver can either return the conflicting assignment $(B_1 \wedge B_2 \wedge B_3 \wedge B_{64} \wedge B_{73})$, or it can derive and return the learned UTVPI constraint $(x+y \leq 2)$ along with the learned clause $(B_1 \wedge B_2 \wedge \neg B_8)$. The latter is preferable since it minimizes future invocations of the ILP solver

Theorem 1. A given system of satisfiable transitively-closed UTVPI constraints together with a set of non-UTVPI linear constraints is *equi-satisfiable* with a subset of those UTVPI constraints sharing both variables with variables in the non-UTVPI system and the set of non-UTVPI constraints.

Proof: Chvátal [9] showed that cutting planes provide a canonical way of proving that every integral solution of a given system of linear inequalities satisfies another given inequality. Therefore, a cutting plane proof of unsatisfiability is to prove that there exists a sequence of inequalities such that their non-negative combination results in inequality $0x \leq -1$. Thus, since we know that the set of UTVPI constraints is satisfiable, the proof of unsatisfiability, if any, should include at least one of the newly added non-UTVPI constraints. In order to yield $0x \leq -1$ from combining linear constraints with those non-UTVPI constraints, it is sufficient to only consider those constraints among the UTVPI constraint that share variables with them knowing that no new constraint can be generated by combining only UTVPI constraints because they are transitively closed. Therefore, the set of non-UTVPI constraints together with those UTVPI constraints which share variables with them is equi-satisfiable with all UTVPI and non-UTVPI constraints. □

To utilize this theorem, the non-UTVPI solver returns the variables associated with its active constraints to the UTVPI solver which uses them to select the subset of UTVPI constraints that must be checked for consistency with the active non-UTVPI constraints. In general, the number of UTVPI constraints that are passed to the ILP solver using this "filter" is much smaller than the total number of active UTVPI constraints, and that directly contributes to much better performance by the ILP solver. Using our running example, the constraints that must be checked by the ILP solver are highlighted in Figure 5.

If the constraints passed to the ILP solver are found to be consistent, the process terminates with a satisfying solution to the original formula. If, on the other hand, the ILP solver finds the constraints to be inconsistent, it must communicate this fact to the Boolean/UTVPI solver which, in turn, must find another solution. This can be done in a manner similar to that used by the UTVPI solver to communicate unsatisfiability to the Boolean solver, namely by returning a set of conflicting indicator variables. This is sufficient, but can be quite inefficient. This is illustrated for our example in Figure 5. After detecting inconsistency, the non-UTVPI solver returns the conflicting assignment $(B_1 \wedge B_2 \wedge B_3 \wedge B_{64} \wedge B_{73})$ to the Boolean solver which uses it to perform conflict analysis and backtracking. Upon finding another satisfying solution, e.g., $(\neg A_1 \wedge \neg A_2 \wedge \neg A_3 \wedge A_4 \wedge B_1 \wedge B_2 \wedge B_3 \wedge B_{63} \wedge B_{73})$, the non-UTVPI solver is invoked again and returns with another conflicting assignment. In fact, this iteration will be repeated twelve times exhausting all possible satisfying assignments to constraints ω_7 and ω_8 before the search process backtracks and reverses an assignment to an A variable.

To reduce the calls to the ILP solver, the non-UTVPI solver can perform its own "intelligent" conflict analysis before returning to the Boolean solver. The goal of this analysis is to derive, using cutting planes, a UTVPI constraint that is implied by the current conflicting assignment. Such a constraint can then be returned to the Boolean/UTVPI solver to help confine the "bactracking" iterations inside that solver. For our example, this is accomplished by combining the non-UTVPI constraint $u + v - 4x - 4y = 0$ with the UTVPI constraint $u + v \leq 11$ yielding $4x + 4y \leq 11$ which is tightened to $x + y \leq 2$. Assigning a fresh indicator variable B_8 to this learned constraint, it is returned to the Boolean/UTVPI solver along with the associated learned clause $(B_1 \wedge B_2 \wedge \neg B_8)$ (see Figure 5). Upon learning this clause, the Boolean/UTVPI solver would eventually know that either A_1 or A_2 should be true by checking all combinations of B_{6i} and B_{7j} without referring to the non-UTVPI solver and those would be the only satisfying solutions to the problem.

As it is obvious, this process will result in an strictly more powerful learned clause than that of the naive method explained earlier and would considerably help the search algorithm by pruning its infeasible search space more efficiently.

5 Implementation and Experimental Results

The UTVPI solver was implemented within our ARIO SMT solver [3]. ARIO is built on top of MiniSAT [12] and inherits its strategy for random restarts, VSIDS and clause removal. For the Simplex-based ILP solver, we used XPRESS-MP [10] and all experiments were conducted on a Pentium-IV 2800MHz machine with 1 GB of RAM running Linux.

To evaluate our method, we experimented on formulas obtained from the Wisconsin Safety Analyzer (WiSA) project [14]. These instances are concerned with finding API-level exploits by introducing a framework to model low-level details of API's, and adopting an automatic technique based on bounded, infinite-state model checking. The benchmarks include both satisfiable and unsatisfiable instances and incorporate UTVPI and non-UTVPI constraints combined with Boolean connectors. Un-interpreted functions were initially eliminated using Ackermann's technique [1] and the satisfiability of the resulting formula was tested using different hybrid SAT solvers. Characteristics of the benchmarks used in this evaluation are displayed in Table 1.

Table 1. WiSA benchmarks

benchmark	**Result**	**Variables int/bin**	**Constraints**		
			CNF	**UTVPI**	**non-UTVPI**
s-20-20	SAT	60/973	864	778	6
s-20-30	SAT	60/973	864	778	6
s-20-40	UNS	60/973	864	778	6
s-30-30	SAT	80/1393	1244	1128	6
s-30-40	SAT	80/1393	1244	1128	6
xs-20-20	SAT	82/1116	1046	959	6
xs-20-30	SAT	82/1116	1046	959	6
xs-20-40	UNS	82/1116	1046	959	6
xs-30-40	SAT	112/1606	1516	1399	6

The results of running ARIO on the benchmarks of Table 1 are displayed in Table 2. This table also includes CPU run-times obtained from [28] for solving the same instances using the parametrized solution bounds method along with

Table 2. Run-times (in sec.) of ARIO, UCLID and ICS on WiSA benchmarks

benchmark	UCLID time[a]	ICS time	ARIO time		
			UTVPI	non-UTVPI	total
s-20-20	8.78	0.25	0.17	0.01	0.26
s-20-30	9.50	0.37	0.32	0.01	0.61
s-20-40	4.50	286.84	2.77	0.01	5.05
s-30-30	20.89	1.64	0.28	0.01	0.45
s-30-40	19.21	7.41	1.21	0.01	2.06
xs-20-20	26.03	17.77	0.35	0.02	0.57
xs-20-30	21.42	1482.80	0.1	0.01	0.23
xs-20-40	14.18	>3600	173.9	0.01	276.43
xs-30-40	33.22	>3600	1.88	0.06	3.01

[a] from [28] and adjusted to CPU speed of 2.8 GHz

UCLID and also the timings of ICS version 2.0c [13] for the same problems translated by ARIO before applying Ackermann conversion. The hybrid method of ICS is based on lazy, online integration of a non-clausal SAT solver with an incremental, backtrackable constraint engine. The approach adopted in their constraint engine is an extension of Nelson's version of Simplex algorithm where equalities and disequalities are added to a Simplex tableau incrementally. Integer constraints are processed, in the same framework with the addition of cutting planes to preserve their discreteness. For details of their method, the reader is referred to [27].

The clear advantage of ARIO method over the online method of ICS is mainly due to its strategy of separating UTVPI and non-UTVPI constraints, its unique strategy for refinement and more efficient handling of integer constraints.

Comparing to the pre-processing and solution-bound approach of UCLID, our method outperformed on all satisfiable instances due to its more effective learning and on-demand collaboration strategy of its solvers. The cases that UCLID performed better, i.e. mainly small *unsatisfiable* instances, are the ones in which all the transitivity constraints could be efficiently pre-encoded into the SAT solver in the form of CNF clauses, effectively shifting all the search work into the SAT solver rather than sharing it with the slower UTVPI solver as in our method. However, it is important to remember the high dependence of such methods on having a low number of separation and non-separation constraints that cause the solver to slow down exponentially as the number of constraints increase[1]

As demonstrated in Table 3, adopting our intelligent refinement method based on cutting plane theory resulted in a considerable decrease in calls to

[1] We could not obtain the version of UCLID supporting integer non-separation constraints from its developers and therefore our information on its performance is limited to what they reported in [28] and the characteristics of their algorithm.

Table 3. Running ARIO on WiSA benchmarks. Number of runs is the number of times a solution to SAT/UTVPI problem was found to be inconsistent by non-UTVPI engine

benchmark	number of conflicts			Number of runs	
	total	in UTVPI	in Cutting Planes	with Cutting Planes	no Cutting Planes (time)
s-20-20	1111	1057	6	10	84 (0.66 s)
s-20-30	3172	3009	12	8	2066 (15.36 s)
s-20-40	30611	30418	3	1	time-out
s-30-30	1500	1436	2	1	447 (10.17 s)
s-30-40	7631	7281	29	11	273 (7.07 s)
xs-20-20	877	811	11	17	160 (2.09 s)
xs-20-30	396	388	3	1	318 (3.62 s)
xs-20-40	748710	746239	3	1	time-out
xs-30-40	3739	3596	18	16	255 (8.01 s)

non-UTVPI solver and in some cases enormous speed-ups. This table also depicts the number of conflicts due to inconsistencies detected among UTVPI constraints that are all detected online rather than being delayed until the search is complete. Large numbers of such conflicts in these problems makes their online handling vital to the efficiency of the overall algorithm. The number of conflicts detected by cutting planes refers to those conflicts that would have needed a call to the non-UTVPI solver to be detected.

6 Related Work, Conclusions and Future Work

In this paper, we presented a new scalable, cooperative and intelligent method for solving decision problems involving integer linear arithmetic constraints together with Boolean variables. The unique characteristics contributing to the efficiency of our combined method can be summarized as follows:

- Separation of UTVPI and non-UTVPI engines based on the general structure of problems in software and hardware verification applications,
- Online collaborative algorithm for solving UTVPI constraints while the Boolean search proceeds,
- Efficient offline communication between the solvers,
- Intelligent and powerful refinement of inconsistencies by the offline solver.

Tight integration of theory solvers into the SAT solver was also suggested in [16] where any given theory (linear arithmetic for instance) is coupled into the generic DPLL-based propositional solver. Similarly, in our method, the UTVPI constraints are tightly integrated into the SAT solver, but non-UTVPI constraints are handled only after a satisfiable solution to the Boolean/UTVPI problem is found. Our proposed refinement method for inconsistencies in the SAT solutions leads to a considerable reduction in the number of offline calls and effectively prunes the search space of the SAT solver. By not integrating a full-scale linear arithmetic theory into the SAT solver, our approach maintains both efficiency and completeness of the DPLL-based solver.

The use of a modified implication graph to take into account mathematical deductions was also proposed in [18]. In their method, the DPLL-style SAT solver is combined with a constraint solver based on Fourier-Motzkin elimination. The hybrid conflict analysis scheme for finite-domain integer and Boolean variables, introduced in their work, is comparable to the hybrid UTVPI and Boolean learning method adopted in our method and illustrated in Figure 4.

In [6], a layered decision procedure for the satisfiability of linear arithmetic logic, implemented in MathSAT, is presented. MathSAT is mainly organized in a layered hierarchy of solvers in increasing solving capabilities. Similar to our method, MathSAT also differentiates between Difference Logic (DL) constraints of the form $x - y \sim c$ (a special class of UTVPI constraints) and general linear arithmetic constraints. The former is solved using a Bellman-Ford algorithm and the latter by a simplex-based solver. Integer constraints are handled using branch-and-bound for small problems or Omega Test [26] as the last resort. Unlike MathSAT, in our method the UTVPI constraints are tightly integrated into the SAT solver and by adopting an incremental approach for solving the UTVPI problem and taking advantage of the property introduced in Theorem 1, the integer problem becomes considerably smaller, effectively eliminate the need to the computationally-expensive Omega Test.

Compared to these methods, our method is more robust and reliable as it preserves the completeness of the procedure while maintaining a high level of efficiency. As we learn more about the trade-offs involved, we will be able to develop effective integration strategies that outperform individual techniques.

One promising direction of future research involves improving the efficiency of the collaboration between the SAT solver and its UTVPI engine and also extending the cutting plane techniques described above to add more refinement power in order to minimize the number of inconsistent solutions. Since in most verification applications, the integer solvers are the bottleneck in the SAT-based algorithms, the methods described in this paper when combined with other ILP methods provide a viable option. Specifically, the independent characteristics of our method together with its ability to collaborate with the SAT solver efficiently, makes it more practical for large problems and more amenable to parallelization.

Acknowledgement

This work was funded in part by the National Science Foundation (NSF) under ITR grant No. 0205288. We also would like to thank Vinod Ganapathy for providing us with WiSA benchmarks.

References

[1] W. Ackermann, "Solvable Cases of the Decision Problem," North-Holland, Amsterdam, 1954.

[2] T. Amon, G. Borriello, T. Hu, and J. Liu, "Symbolic Timing Verification of Timing Diagrams Using Presburger Formulas," *DAC*, pp 226-231, 1997.

[3] ARIO SMT Solver, http://www.eecs.umich.edu/~ario

[4] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowics, and R. Sebastiani, "A SAT-Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions," *CADE*, pp. 193-208, 2002.

[5] C. Barrett, D. Dill, and J. Levitt , "Validity Checking for Combinations of Theories with Equality," *FMCAD, LNCS 1166*, pp. 187-201, 1996

[6] M. Bozzano, R. Bruttomesso, A. Cimatti, T.A. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani, "An Incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic," *TACAS*, pp 317-333, 2005.

[7] R. Brinkmann and R. Drechsler, "RTL-datapath Verification Using Integer Linear Programming," *VLSI Design*, pp 741-746, 2002

[8] J. A. Brzozowski and C.J.H. Seger, "Asynchronous Circuits," *Springer*, 1994.

[9] V. Chvátal, "Edmonds polytopes and a hierarchy of combinatorial problems," Discrete Math. vol. 4 pp. 305-337, 1973.

[10] Dash Inc., XPRESS-MP 15.25.03, http://www.dashoptimization.com.

[11] D.L. Detlefs, G. Nelson, and J.B. Saxe, "Simplify: A Theorem Prover for Program Checking," Tech Report HPL-2003-148, HP Labs, 2003.

[12] N. Eén and N. Sörensson, "An Extensible SAT-solver," SAT, pp. 502-508, 2003.

[13] J.C. Filliatre, S. Owre, H. Rueß and N. Shankar, "ICS: Integrated Canonizer and Solver," *CAV*, pp. 246-249, 2001.

[14] V. Ganapathy, S.A. Seshia, S. Jha, T.W. Reps, R.E. Bryant, "Automatic Discovery of API-Level Exploits," *ICSE*, 2005

[15] V. Ganesh, S. Berezin, and D.L. Dill, "Deciding Presburger Arithmetic by Model Checking and Comparisons with Other Methods," *FMCAD*, pp. 171-186, 2002.

[16] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast Decision Procedures," *CAV*, pp. 175-188, 2004.

[17] ILOG CPLEX, http://www.ilog.com/products/cplex.

[18] M.K. Iyer, G. Parthasarathy and K.-T. Cheng, "Efficient Conflict-Based Learning in an RTL Circuit Constraint Solver," *DATE*, pp 666-671, 2005.

[19] J. Jaffar, M. Maher, P. Suckey, and R. Yap, "Beyond Finite Domains," *Workshop on Principles and Practice of Constraint Programming*, 1994.

[20] D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman, "Abstraction-based Satisfiability Solving of Presburger Arithmetic," *CAV*, pp.308-320, 2004

[21] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. on Computers*, vol. 48(5), pp. 506-521, 1999.

[22] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," DAC, pp. 530-535, 2001.

[23] L. de Moura and H. Rueß, "An Experimental Evaluation of Ground Decision Procedures," *CAV* pp. 162-174, 2004.

[24] G. Nelson, and D.C. Oppen, "Simplification by Cooperating Decision Procedures," ACM Trans. on Programming Languages and Systems vol. 1, pp 245-257, 1979.

[25] M. Presburger, "Uber die Vollstandigkeit eines Gewissen Systems der Arithmetik Ganzer Zahlen, in Welchem die Addition als Einzige Operation Hervortritt," Comptes-rendus du premier congres des mathematiciens des pays slaves, 395: pp 92-101, 1929.

[26] W. Pugh, "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis," *ACM conf. on Supercomputing*, pp. 4-13, 1991.

[27] H. Rueß, and N. Shankar, "Solving Linear Arithmetic Constraints," SRI International Tech Report CSL-SRI-04-01, January 2004.

[28] S. Seshia and R. Bryant, "Deciding Quantifier-Free Presburger Formulas Using Parameterized Solution Bounds," *LICS*, pp. 100-109, 2004.

[29] R. Shostak, "Deciding Combination of Theories," *Journal of the ACM* vol. 31 pp. 1-12, 1984.

[30] R. Shostak, "Deciding Linear Inequalities by Computing Loop Residues," *Journal of the ACM*, vol. 28(4) pp. 769-779, 1981.

[31] O. Strichman, S.A. Seshia, and R.E. Bryant " Deciding Separation Formulas with SAT", *CAV*, pp 209-222, 2002.

[32] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A First Step Towards Detection of Buffer Overrun Vulnerabilities," *Network and Distributed System Security Symposium*, Internet Society, 2000.

[33] J.M. Wilson, "Compact Normal Forms in Propositional Logic and Integer Programming Formulations," *Computers and Operation Research*, pp. 309-314, 1990.

DPVIS – A Tool to Visualize the Structure of SAT Instances

Carsten Sinz and Edda-Maria Dieringer

Symbolic Computation Group, WSI for Computer Science,
University of Tübingen, 72076 Tübingen, Germany
{sinz, dieringe}@informatik.uni-tuebingen.de

Abstract. We present DPVIS, a Java tool to visualize the structure of SAT instances and runs of the DPLL (Davis-Putnam-Logemann-Loveland) procedure. DPVIS uses advanced graph layout algorithms to display the problem's internal structure arising from its variable dependency (interaction) graph. DPVIS is also able to generate animations showing the dynamic change of a problem's structure during a typical DPLL run. Besides implementing a simple variant of the DPLL algorithm on its own, DPVIS also features an interface to MiniSAT, a state-of-the-art DPLL implementation. Using this interface, runs of MiniSAT can be visualized—including the generated search tree and the effects of clause learning. DPVIS is supposed to help in teaching the DPLL algorithm and in gaining new insights in the structure (and hardness) of SAT instances.

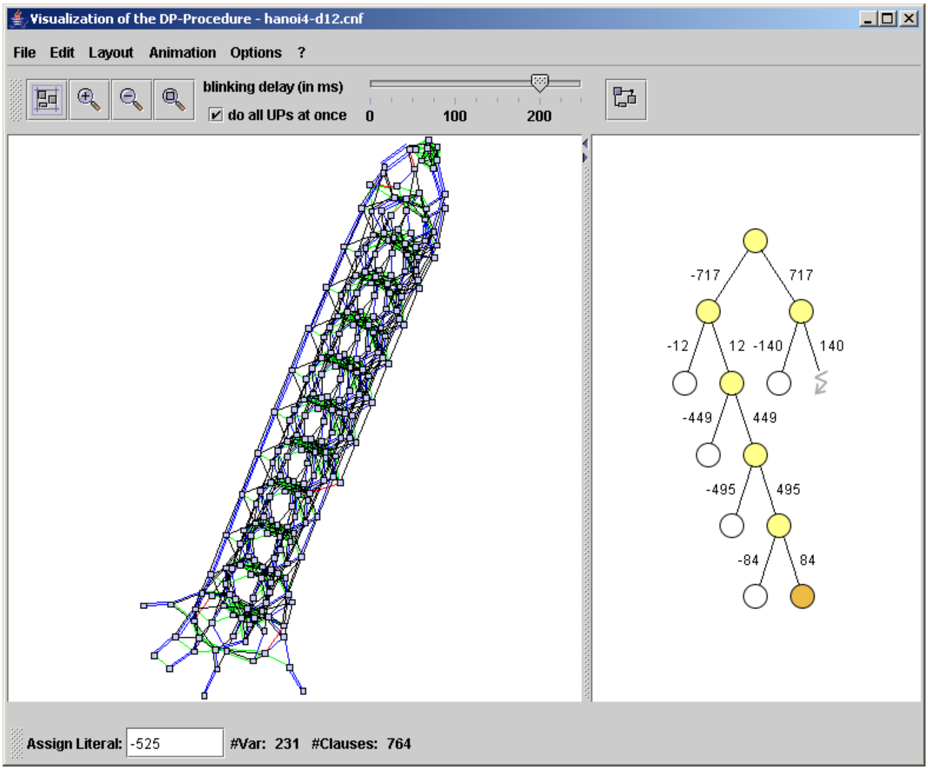
1 Introduction

Although SAT is an NP-complete problem, there are many real-world instances that can be solved surprisingly fast by modern (mainly DPLL-based) solvers. The typical explanation for this phenomenon that can be found in the literature is that those instances are equipped with some kind of internal (and sometimes hidden) "structure" that makes these problems tractable. The term "structure", due to its vagueness, leaves much room for interpretation, though, and it remains unclear how this structure manifests itself and how it could be exploited. We have therefore proposed a visualization approach [1], that is supposed to deliver some hints on why solving a particular instance is hard or easy.

Our approach is based on the problem's *(variable) interaction graph* [2], which is supposed to appropriately reflect (at least part of) the problem's structure. In a SAT instance given by a set S of clauses over a set X of propositional variables, the interaction graph's vertices are the instance's propositional variables, and variables occurring together in any clause of S are connected by an edge. Thus, if two variables are connected, this indicates that assigning a truth value to one of the connected variables has the potential to determine the truth value of the other (thus they *interact*).

DPVIS (see Fig. 1) reads problems in DIMACS CNF format and shows their variable interaction graph. The interaction graph is laid out using algorithms that are known to reflect graph clustering and symmetry especially well [3]. Moreover, DPVIS visualizes changes of the interaction graph during the run of a DPLL algorithm. This includes visualizing the effects of unit propagation (Boolean constraint propagation) on the interaction graph as well as showing the search tree generated by the DPLL algorithm.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 257–268, 2005.



blinking delay (in ms)

☑ do all UPs at once

Assign Literal: -525 #Var: 231 #Clauses: 764

Fig. 1. DPVIS tool showing a visualized SAT instance (**hanoi4** of the DIMACS Benchmark Collection): The variable interaction graph is shown on the left and a manually generated, partial search tree on the right

We expect DPVIS to be a useful tool for examining the structure of SAT instances—especially in generating hypotheses on what makes an instance tractable. Moreover, we believe that DPVIS can be a valuable tool in teaching the DPLL algorithm. As DPVIS can also display learned clauses, non-chronological back-jumping, and search restarts, it encompasses most features that can be found in a modern implementation of the DPLL procedure.

2 Theoretical Background

2.1 SAT Instances as Graphs

To transform a SAT instance represented as a set S of clauses into a (directed or undirected) graph $G = (V, E)$ with vertex set V and edge set E a number of methods have been suggested [2, 4, 5, 6]. We have decided to use a variant of Rish and Dechter's *interaction graphs*[1] for our visualization, as they put an emphasis on variable dependencies.

[1] Interaction graphs are also known as *primal graphs*. Slater [6] calls them *co-occurrence-of-variables-graphs*.

The interaction graph is an undirected graph, where the vertex set V is the set of variables of S and $\{x, y\} \in E$ if and only if there is a clause $c \in S$ that contains both variables x and y. Note, that this is a lossy representation, as the signs of literals in clauses are ignored. For the purpose of structural analysis and visualization we consider this not a disadvantage, however.

For our visualization experiments we have decided to use a refined variant of interaction graphs, in which 2-clauses (i.e. clauses containing exactly two literals) are represented as visually emphasized directed or undirected arcs in the graph. We make the following distinction:

1. A clause with two positive literals is shown as a red, double-ended arrow.
2. A clause with one positive and one negative literal, say $(\neg x \vee y)$, is written as a blue arrow from variable x to variable y, indicating the implication direction.
3. A clause with two negative literals is written as a green, double-ended arrow.

All other clauses are displayed without highlighting, i.e. as black edges between the involved variables. Moreover, we allow duplicate edges in our graphs; thus, e.g., two clauses both involving variables x and y result in a double edge between the corresponding vertices.

Special treatment of 2-clauses is motivated by their importance for tractability: A problem consisting only of 2-clauses is solvable in linear time [4]. Moreover, experiments with random instances from the $(2 + p)$-model (problems with a fraction of p 3-clauses and $(1 - p)$ 2-clauses) indicate that random instances with up to 40 percent of 3-clauses (i.e. $p \leq 0.4$) might be computationally tractable [7].

2.2 Graph Layout

The *graph layout* or *graph drawing* problem consists of generating a geometric representation of a graph in two or three dimensions. Nodes have to be positioned in the Euclidean space while optimizing certain layout properties like minimal number of edge crossings, uniform edge length, reflection of inherent symmetry, etc [3, 8].

Different algorithms are available for graph layout, a prominent one being the *spring embedder model* of Eades [9] or its close relative, the *force-directed placement* algorithm of Fruchterman and Reingold [3]. Both are known to produce layouts that reflect symmetry convincingly.[2] The physical model used by the spring-embedder assumes metal springs of a certain length attached between each pair of connected nodes. The springs impose attractive and repellent forces on the nodes depending on their current distance in the layout. The layouter attempts to minimize the sum of all affecting forces by iteratively repositioning the nodes.

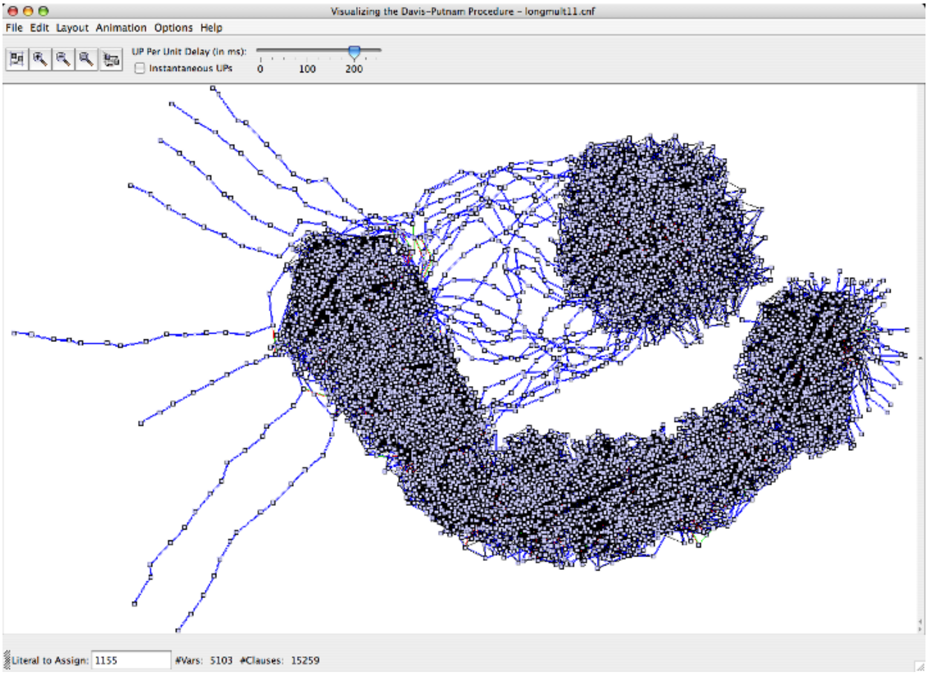
DPvis builds upon the commercially available graph layout package yFiles of yWorks (`http://www.yworks.com`), from which it uses the *Organic Layouter* and *Smart Organic Layouter*, both of which implement force-directed placement algorithms.

[2] We have considered other existing layouts as well, like hierarchical or orthogonal layout, but found them not being equally suitable for our purpose.

3 DPVIS Functionality

Upon start-up, DPVIS reads a SAT instance in DIMACS CNF format from a file, performs unit propagation, and displays the instance's variable ineraction graph with variables (the graph's nodes) placed randomly (on the left part of the screen) and an empty DPLL search tree (on the right). The user can then choose a graph layouter and DPVIS computes a visually more attractive layout. Thereafter, variables can be set to *true* or *false* (by clicking on the graph's nodes with either the left or right mouse button), or an automatic DPLL run can be started. In the first case, after setting a variable, unit propagation is automatically initiated.

A detailed description of all available display and animation options is given in the following paragraphs.


Visualizing the Davis–Putnam Procedure – longmult11.cnf
File Edit Layout Animation Options Help
UP Per Unit Delay (in ms):
0
100
200

□ Instantaneous UPs

Literal to Assign: 1155 #Vars: 5103 #Clauses: 15259

Fig. 2. Larger layout example: Bounded model-checking instance **longmult11** (see Sec. 4 for a reference) with 5103 variables and 15259 clauses laid out using DPVIS' Smart Organic Layouter

3.1 Visualizing Internal Structure

To display the static problem structure and show the effects of setting individual variables and subsequent unit propagation, DPVIS offers these features:

- **Two Different Layout Algorithms:** Graph layouts can be computed using two different force-directed layout algorithms, each equipped with a set of additional parameters, e.g. for controlling preferred edge length or graph compactness.

- **Zooming the Interaction Graph:** After the layout is computed, the user can zoom into the interaction graph to focus on an area of special interest. The user may also search for variables and center the view on them, or optimally fit the graph into the available display area.
- **Setting Variables to *true* resp. *false*:** By clicking on a variable, the user can set a variable to *true* (by clicking the left mouse button) or *false* (right mouse button). Each setting of a variable also extends the search tree by a new leaf.
- **Performing Unit Propagation:** Unit propagation is automatically initiated after setting of a variable: first, all variables affected by unit propagation are highlighted; then they are removed one by one (or at once, if this option is selected) and the interaction graph is updated accordingly, resulting in an animated view of unit propagation.

An example of a typical layout obtained with DPvis is displayed in Fig. 2.

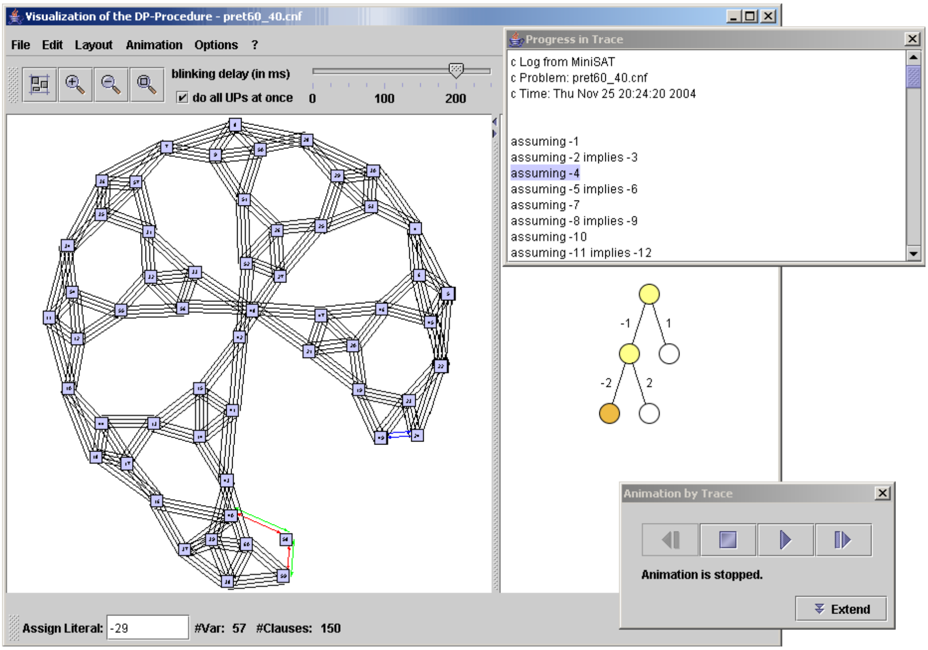
3.2 Animating DPLL Runs

DPvis also allows generating and visualizing complete runs of the DPLL procedure. There are three possibilities to generate such a run. They differ in the way in which the case-splitting literal L is selected in the DPLL algorithm (see Fig. 3).

1. **Manual Selection of Case-Distinction Variable:** The user selects the case-splitting variable by clicking on a node in the variable interaction graph or by entering the variable index into a field at the bottom of the screen. This option delivers enough flexibility for the user to experiment with his own (manually generated) variable selection heuristics, e.g. one that tries to generate independent subproblems.
2. **Simple Variable Selection Heuristics:** DPvis also implements two simple variable selection heuristics: one that randomly selects the case-splitting variable, and one that is based on counting literal occurrences: the variable with the maximal number of occurrences (making no distinction between positive and negative occurrences) is selected.
3. **Interface to Modern DPLL Implementation:** An interface to external SAT-solvers is also contained in DPvis. This allows DPvis to read traces of DPLL runs from

```
boolean DPLL(ClauseSet $S$)
{
  while ($S$ contains a unit clause $\{L\}$) {
    delete clauses containing $L$ from $S$       // unit-subsumption
    delete $\overline{L}$ from all clauses in $S$          // unit-resolution
  }
  if ($\emptyset \in S$) return false                    // empty clause?
  if ($S = \emptyset$) return true                     // no clauses?
  choose a literal $L$ occurring in $S$           // case-splitting on $L$
  if (DPLL($S \cup \{\{L\}\}$) return true              // first branch
  else if (DPLL($S \cup \{\{\overline{L}\}\}$) return true        // second branch
  else return false
}
```

Fig. 3. Pseudo-code of the basic DPLL algorithm



```
c Log from MiniSAT
c Problem: pret60_40.cnf
c Time: Thu Nov 25 20:24:20 2004


assuming -1
assuming -2 implies -3
assuming -4
assuming -5 implies -6
assuming -7
assuming -8 implies -9
assuming -10
assuming -11 implies -12
```

Animation is stopped.

Assign Literal: -29 #Var: 57 #Clauses: 150

Fig. 4. DPVIS-interface to MiniSAT. Program traces (top right window) produced by MiniSAT are read in by DPVIS, and can be played back. Play-back is controlled by an animation console (bottom right window)

solvers like zChaff or MiniSAT[3]. Currently, the only interface available is that to MiniSAT (see Fig. 4). Using this interface, DPLL traces containing information about case-splitting, unit propagation, learned clauses and back-jumps can be animated and analyzed with DPVIS. The user can choose between playback mode, in which—starting with an initial interaction graph layout—the program trace is presented to the user (resembling a movie). Alternatively, the user can employ single-step mode, in which after each step he or she may compute a new graph layout.

Independent of which mode was used to generate the DPLL search tree, the following additional options are available:

- **Free navigation in the search tree:** At each point in the DPLL program trace, the user can stop the trace and navigate to any point in the thitherto existing search tree.
- **Re-compute layout at any time and each node of the search tree:** When the playback of the animation trace is stopped (or interrupted), different layout algorithms and parameter settings may be applied either to the current state (see Fig. 5) or (by combining it with the search tree navigation feature) to already visited search nodes. Different states during search can thus easily be compared.

[3] SAT-solvers that are intended to interface with DPVIS have to be slightly modified in order to output the appropriate program traces.

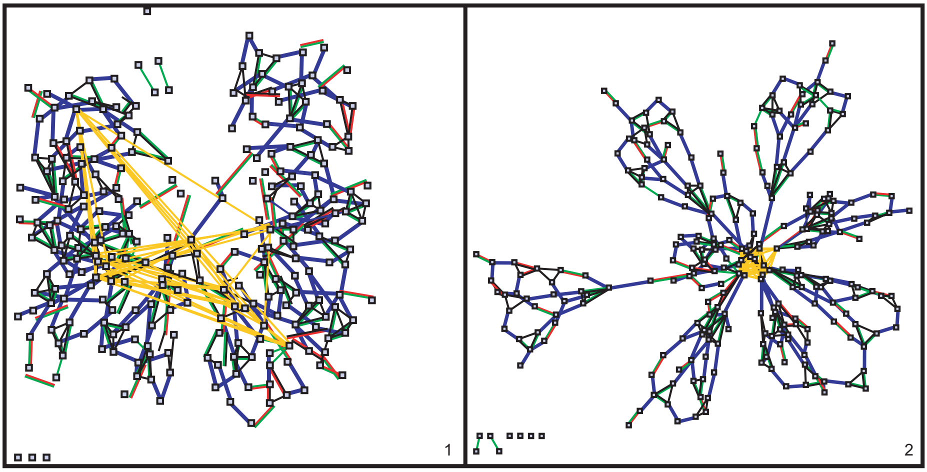


Fig. 5. At each point during play-back of a DPLL trace, the layout of the interaction graph may be re-computed: after having learned some clauses (left diagram, yellow arcs; instance **ssa0432-003** of the DIMACS Benchmark Collection), re-computing the graph layout shows the locality of the learned lemmas (right diagram)

4 Two Sample Applications

We now briefly present two examples, how DPvis can effectively be employed. The first is concerned with the analysis of DPLL search spaces, the second deals with so-called *top-level assignments*.

4.1 Comparing Search Spaces

The intention of this experiment is to compare search spaces resulting from random problem instances with search spaces generated by "real-world" problems. We use the following SAT instances for our experiments:

longmult1: encoding of a bounded model checking (BMC) problem (equivalence of output bit 0 of two hardware multiplier designs)

uuf50-0188: unsatisfiable uniform random 3-SAT instance with 50 variables and a clause-variable-ratio of $\alpha = 4.36$ (i.e. near the satisfiability threshold)

ssa2670-141-d7: instance from test-patterngeneration(checks for *single-stuck-at* fault), where seven randomly selected variables have been fixed (to random Boolean values in order to reduce the search space to a size comparable with the other instances)

bw_large.b-d8: encoding of an AI planning problem, with eight randomly selected variables fixed to random Boolean values (as with ssa2670-141-d7)

The **longmult1**-instance can beobtainedfrom`www.cs.cmu.edu/`~{}`modelcheck /bmc.html`, all other instances can be downloaded from `www.satlib.org`. All instances are unsatisfiable, additional statistical information about these problem instances can be found in Table 1.

Table 1. Characteristic numbers for some SAT instances (all instances unsatisfiable), including MiniSAT results (number of conflicts and decisions) and balancedness-coefficients

Instance	#Vars	#Clauses	#Confl.	#Dec	B-Coeff.
longmult1	631	1611	5	20	3.7021
uuf50-0188	50	218	145	186	1.0611
ssa2670-141-d7	753	1619	41	253	7.3231
bw_large.b-d8	889	9949	4	19	3.0603

The table's columns show—from left to right—the instance name, the number of variables and the number of clauses occurring in the problem (after having performed unit propagation), the number of conflicts and decisions produced during a MiniSAT run, and the B-coefficient, which is an indicator of the balancedness of the search tree of the instance (as generated by MiniSAT). More exactly, for a search tree T the B-coefficient B is defined by $B := h/\lceil \mathrm{ld}(c) \rceil$, where h is the height of the search tree (the length of the longest path from the root), c is the number of conflicts produced by MiniSAT (i.e. the number of leaves in the search tree), and $\lceil \mathrm{ld}(c) \rceil$ is the dual logarithm of c rounded up to the next larger integer, i.e. the height of a balanced tree with c nodes. Thus, a B-coefficient of 1 indicates a balanced tree, whereas a large coefficient stands for an unbalanced, degenerate tree.

The search spaces for all four problem instances are shown in Fig. 6. The random 3-SAT instance (Fig. 6-2) reveals an almost balanced search tree, whereas the other, "real-world" instances (1, 3, 4) possess more or less degenerate trees. This observation is also reflected by the B-coefficient of the real-world instances being much higher than that of the random 3-SAT instance.

We suppose that hard SAT instances have low B-coefficients whereas easy instances possess high ones. Further assuming that the search space is uniform (in the sense of having similar B-coefficients in different parts), the B-coefficient may be used to estimate the run-time of a SAT solver after having processed only a small fraction of the search space.

We expect that by making experiments with DPVIS other notions similar to that of the B-coefficient might come up in the future, hopefully allowing for a better distinction between hard and easy problem instances.

4.2 Analyzing the Effect of Top-Level Assignments

Eén and Sörensson introduced the notion of *top-level assignments* for unit-clauses that are learned during search [10]. Each top-level assignment (TLA) fixes the value of a variable for the whole instance and thus indicates that in each solution of the instance the TLA-variable must have that fixed value.[4]

In a further experiment with DPVIS, we compared the number of TLAs produced by different SAT instances during a MiniSAT run. Some results of these experiments are presented in Table 2.

[4] The notion of a top-level assignment is closely related to that of a *backbone variable*[11].

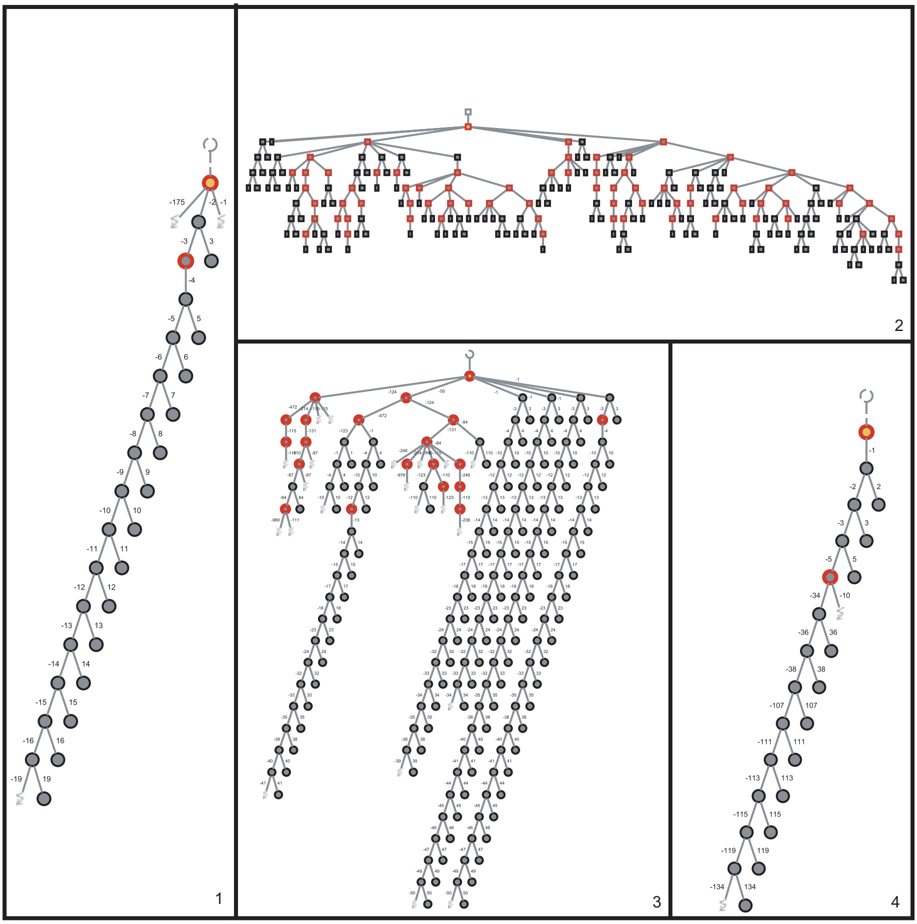


Fig. 6. Search trees for different problem instances: 1: bounded model checking (longmult1); 2: random 3-SAT (uuf50-0188); 3: test-pattern generation (ssa2670-141_d7); 4: planning problem (bw_large.b-d8)

Table 2. Number and percentage of top-level assignments (TLAs) for some SAT instances

Instance	#Vars	#Clauses	#TLAs	#TLA/#Vars
longmult1	631	1611	151	23.93%
uuf50-0125	50	218	13	26.00%
ssa2670-141-d7	753	1619	336	44.62%
bw_large.b-d8	889	9949	233	26.21%

Although the number of top-level assignments is considerable in all cases, a more elaborate study with more problem instances from each problem class showed that the number of TLAs is typically much higher for real-world problems than for random 3-SAT problems (29.5% vs. 11.0% in our experiments).

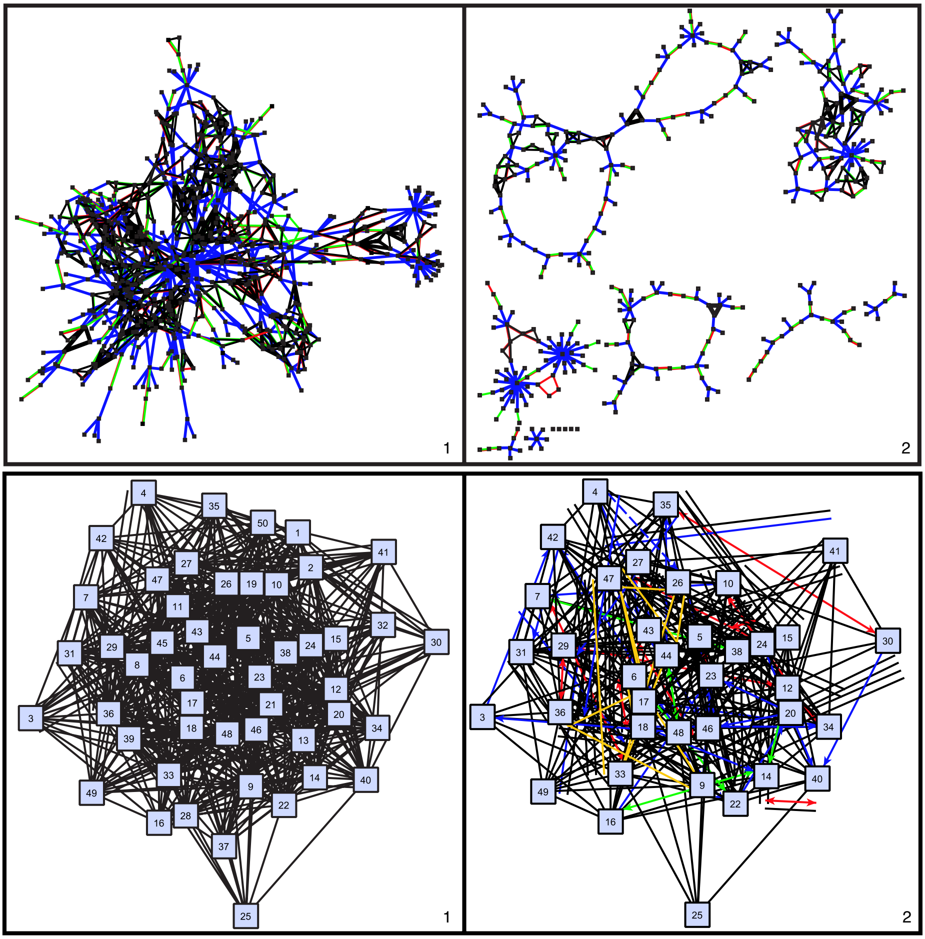


Fig. 7. Comparing interaction graphs before (on the left) and after (on the right) having added TLAs (top-level assignments): Real-world instance **ssa2670-141-d7** (shown on top) reveals a considerable simplification and decomposition into independent components after having added TLAs. The random 3-SAT instance **uuf50-0125** (shown below) also possesses a considerable amount of TLAs, but exhibits no such simplification

Comparing (using DPVIS) the interaction graphs of the original instances with those where the detected TLAs were added, revealed a considerable simplification for the real-world instances, whereas the structure of random 3-SAT instances remained mainly unchanged (see Fig. 7).

5 Conclusion and Related Work

We presented DPVIS, a tool to visualize the structure of SAT instances and to display DPLL search trees. We see a twofold purpose for our visualization tool: First, it may help in building hypotheses on why problem instances are hard or easy. We have indicated in Sec. 4 how experiments in this direction might look like. Second, we suggest using DPVIS in teaching the basic DPLL algorithm and recent extensions of it like clause learning and restarts. By having an integrated view on both the problem structure (via its variable interaction graph) and the search tree, students can obtain a good intuition how the DPLL method works.

Future experiments should include the analysis of an instance's component structure (resp. its decay into independent components) and the examination of long implication chains, as suggested in [1]. Moreover, in order to handle large graph layouts more conveniently, it would be desirable to have additional tools, e.g. for grouping sets of variables or merging them into a single node.

The only work on visualization of SAT instances that we are aware of is that of Slater [6] and preliminary work by Selman [12]. Slater uses different graph translation techniques (interaction graph, co-occurrence of literals and further ones) and visualizes the resulting graphs with the GraphVis software package from AT&T. However, the hierarchical layouter he uses is not as efficient in revealing symmetries as force-directed placement algorithms are. Selman uses a specialized three-dimensional layouter to display variable interaction graphs. In his approach nodes with different degree are placed on different "height levels", and nodes with the same degree are equally distributed on circles growing with the number of nodes that have to be placed on them.

Availability: DPVIS is available both as on-line version (Java Applet) and stand-alone application (Java Application including MiniSAT) from http://www-sr.informatik.uni-tuebingen.de/˜sinz/DPvis.

References

1. Sinz, C.: Visualizing the internal structure of SAT instances (preliminary report). In: Proc. of the 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2004), Vancouver, Canada (2004)
2. Rish, I., Dechter, R.: Resolution versus search: Two strategies for SAT. J. Automated Reasoning **24** (2000) 225–275
3. Fruchterman, T., Reingold, E.: Graph drawing by force-directed placement. Software – Practice and Experience **21** (1991) 1129–1164
4. Aspvall, M., Plass, M., Tarjan, R.: A linear-time algorithm for testing the trurh of certain quantified boolean formulas. Information Processing Letters **8** (1979)
5. Park, T., Van Gelder, A.: Partitioning methods for satisfiability testing on large formulas. Information and Computation **162** (2000) 179–184
6. Slater, A.: Visualisation of satisfiability problems using connected graphs (2004) `http://rsise.anu.edu.au/˜andrews/problem2graph`.
7. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: Determining computational complexity from characteristic 'phase transitions'. Nature **400** (1999) 133–137
8. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.: Algorithms for automatic graph drawing: An annotated bibliography. Computational Geometry **4** (1994) 235–282
9. Eades, P.: A heuristic for graph drawing. Congressus Numerantium **42** (1984) 149–160

10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. of the 6th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2003), Springer (2003) 502–518
11. Singer, J., Gent, I., Smaill, A.: Backbone fragility and the local search cost peak. Journal of Artificial Intelligence Research **12** (2000) 235–270
12. Selman, B.: Algorithmic adventures at the interface of computer science, statistical physics, and combinatorics. In: Proc. of the 10th Intl. Conf. on Principles and Practice of Constraint Programming (CP 2004), Springer (2004) 9–12

Constraint Metrics for Local Search

Finnegan Southey

University of Alberta, Dept. of Computing Science

Abstract. Over the years, a steadily improving series of local search solvers for propositional satisfiability (SAT) have been constructed. However, these solvers are often fragile, in that they have apparently minor details in their implementation that dramatically affect performance and confound understanding. In order to understand and predict the success of differing strategies, various local search *metrics* have been proposed. Many of these metrics summarize properties of the boolean assignments examined during the search. This has two consequences: first, they only capture one side of satisfiability, failing to characterize the behaviour with respect to constraints. Secondly, the boolean requirement limits the applicability of these metrics to more general *constraint satisfaction problems* (CSPs), which can have non-boolean domains.

In response, we present *dual metrics*, derived from existing *primal* (boolean assignment) metrics, that are based on the states of constraints during the search. Experimental results show a strong relationship between the primal and dual versions of these metrics on a variety of random and structured problems. This dual perspective can be easily applied to both SAT and general CSPs, allowing for new insights into the workings of a broad class of local search methods.

1 Introduction

Local search methods for satisfiability (SAT) came to notice with the surprisingly good performance of GSAT, giving rise to a wave of local search research in the subsequent years. A wide variety of algorithms [14, 5, 13, 10, 16, 12, 9, 15] have been proposed over the years, often as the result of extensive experimentation with minor variations. Some of these algorithms are quite fragile, having small and seemingly unimportant details that nonetheless have tremendous impact on performance (e.g the variants of WalkSAT discussed in [10]). This makes it very hard to predict whether a given method will work well, or to *a priori* design features likely to succeed. Where intuition seems somewhat unreliable and formal analysis is difficult, objective, empirical *metrics* can assist the comparison of algorithms and provide insight into their underlying properties.

Aside from improving understanding, metrics can serve as mechanisms for the automatic tuning of parameters [16, 9] or for the selection of algorithms from a collection of methods [8]. They can also offer insights into different classes of problems and their interactions with various methods. SAT has been the primary area of investigation for local search CSP methods, and for their metrics in particular. Most of the proposed metrics characterize the behaviour of assignments to the boolean variables, often featuring functions of Hamming distances. In more general CSPs the domains are not necessarily boolean, or even of uniform size across variables. Domain-values may be ordinal, in

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 269–281, 2005.

which case distance measures are natural, but frequently they are not, and the lack of obvious distance measures leads to *ad hoc* metrics.

We propose two new metrics based on existing measures for SAT. These previously explored metrics are functions of the boolean variable assignments, which we will call the *primal space*. We apply these metrics instead to the *dual space* of constraints. Since constraints are either satisfied or unsatisfied, they form a natural boolean domain, even when primal space is non-boolean. The intuitive interpretation of these metrics from the primal space carry over nicely to the dual space. Any local search method for SAT or general CSPs can be evaluated in terms of these dual metrics. For general CSPs, we gain a set of metrics that we can apply to any local search. In the SAT case, we gain a new perspective, examining algorithms in terms of their exploration of infeasible regions.

We start by explaining a particular set of metrics in detail. We then present the dual versions of these metrics. Experimental results on random and structured SAT instances are used to compare the dual and primal metrics. These results show that the dual versions are in remarkably close agreement with the primal versions and offer a similar ability to predict solver performance. The experiments also expand previous results on primal metrics by demonstrating them on structured instances as well as on random instances. Related metrics from the literature are then explored, showing the relevance of these results by indicating where similar metrics have been used before. We conclude with some discussion on the potential of these metrics.

2 Three Metrics

A previous study of local search metrics identified three useful measures, demonstrating their value on random 3-SAT instances [11]. These measures were: *depth*, *mobility*, and *rate of coverage*. We will describe these metrics in some detail before showing how they can be related the dual space. Since we also present previously unavailable results for these original metrics (i.e. on structured instances), it is worth understanding the originals in some detail.

2.1 Depth

A very natural metric when using an objective function, and one commonly used by experimenters, is the solver's average objective value over the course of the search. The use of an objective function to evaluate neighbours implicitly assumes that solutions are likely to be found near favourable objective values. Under this assumption, one might suppose that the more time spent in regions with favourable objective values, the more likely one is to find solutions. A weaker but safer claim is that spending a lot of time in regions with unfavourable objective values is unlikely to produce solutions.

The simplest measure of what we will call *depth* is the average objective value over all points visited during the search. Practically, this can give rise to a problem. A poor solver may spend a long time searching for a solution before finding one, whereas a good solver may find one very quickly. In measuring depth we are really interested in the long term behaviour, the amount of time spent in promising regions. In hill-climbing searches with random starting points, the searcher frequently starts in poor regions and rapidly descends to much better regions. However, a particularly effective solver may find a solution during, or soon after, this initial descent. Thus, its average behaviour will look quite poor compared to a weaker solver that has a similar initial descent but spends more time searching.

There are several possible corrections one can imagine, but one simple choice is to ignore some fixed number of initial steps d_{skip} to get past the initial descent, and then average over the remaining search. The number of steps to skip can be easily selected by examining several initial descents. In all results reported here, the same skip count $(d_{skip} = 100)$ was used across all solvers after examining their collective behaviour.

$$depth(d_{skip}) = \frac{1}{T - d_{skip}} \sum_{t=d_{skip}+1}^{T} g(\mathbf{a}^{(t)}) \qquad (1)$$

where $g()$ is the objective function used by the solver, or some other objective for which we expect the solver to maintain reasonable values, T is the number of steps taken during the search, and the vectors $\mathbf{a}^{(t)}$ are assignments to the boolean variables. Note that if $T \leq d_{skip}$, the average depth is zero.[1]

2.2 Mobility

Clearly depth alone is not sufficient. One easy way to achieve good average depth is to find a region of favourable values and simply stay there. However, a successful search must explore new regions. Mobility is a measure that detects this tendency. It is simply the average Hamming distance travelled over some fixed interval. For each step t in the search, we compute the Hamming distance $H(\mathbf{a}^{(t)}, \mathbf{a}^{(t+w)})$ between the assignment at step t and the assignment w steps later. Mobility is the average of these distances:

$$mobility(w) = \frac{1}{T - w} \sum_{t=1}^{T-w} H(\mathbf{a}^{(t)}, \mathbf{a}^{(t+w)})$$

where T is the total number of steps in the search.

This metric has the disadvantage of having a single parameter, w, which is the length of the window. Again, it is not hard to select a value for this parameter by looking at average run lengths for solvers and also the maximum possible Hamming distance. Local search solvers which flip only a single variable at each step cannot move more than $min(w, n)$ steps within a window, where n is the number of variables. In all our experiments here, we select the window size to be the same as the number of variables (i.e. $w = n$).

Given that we have a maximum distance, it also makes sense to use a normalized version of this measure:

[1] Comparing depth in solvers with different objective functions (e.g WalkSAT and ESGint) is meaningless unless depth is defined in terms of some common objective function. Therefore, we have measured depth in the same objective function, $g_{UNSAT}()$, for all solvers (i.e. the number of unsatisfied clauses), while the solvers search in their own particular objective.

$$normmobility(w) = \frac{1}{T-w} \sum_{t=1}^{T-w} \frac{H(\mathbf{a}^{(t)}, \mathbf{a}^{(t+w)})}{min(w, n)}$$

We report the normalized version in all experiments.

2.3 Coverage

High mobility ensures that a solver does not simply sit in one place. However, given the windowed nature of the measurement, it is still possible to be trapped in a large basin in the search space. A solver can repeatedly visit a large "circle" of assignments, with a length considerably larger than the window, leading to a false conclusion. One solution is to consider how much of the search space is actually visited by the solver.

This informal notion of *coverage* makes considerable intuitive sense but turning it into a precise measure presents some problems. One possibility is simply to count the number of unique assignments visited. This is a very weak notion for a couple of reasons. First, the number of possible assignments is exponential, so that the actual proportion explored becomes increasingly trivial in any practical application. Second, all visited points could be near each other, so we obtain little information about the global nature of the search.

A more interesting measure can be obtained by considering the Hamming distance between some explored point $\mathbf{a}_e$ and an unexplored point $\mathbf{a}_u$. In particular, we would like information about the distance between a given $\mathbf{a}_u$ and its nearest explored point. This distance can be normalized by dividing by the maximum distance. We call this normalized distance the *gap* of $\mathbf{a}_u$:

$$gap(\mathbf{a}_u) = \frac{1}{n} \min_{\mathbf{a}_e \in \mathbf{A}_e} H(\mathbf{a}_e, \mathbf{a}_u)$$

where $\mathbf{A}_e$ and $\mathbf{A}_u$ are the sets of explored and unexplored points, respectively, and $\mathbf{a}_u \in \mathbf{A}_u$.

If we now consider all unexplored points, the largest gap gives us a notion of how well our space has been covered by our search. If a large gap exists, then some unexplored point is far from any explored point, and we consider this to be poor *coverage*:

$$maxgap = \frac{1}{n} \max_{\mathbf{a}_u \in \mathbf{A}_u} \min_{\mathbf{a}_e \in \mathbf{A}_e} H(\mathbf{a}_e, \mathbf{a}_u)$$

For the intuitive convenience of having a coverage measure that is larger as coverage improves, we define coverage to be the maximum distance (n) minus the maximum gap:

$$coverage = \frac{1}{n}[n - \max_{\mathbf{a}_u \in \mathbf{A}_u} \min_{\mathbf{a}_e \in \mathbf{A}_e} H(\mathbf{a}_e, \mathbf{a}_u)]$$

Unfortunately, computing this value is NP-hard. It is equivalent to computing the *covering radius* of a set of codes [4]. In our research, the following approximation was used. The furthest point from any given assignment, **a,** is its complement, $\bar{\mathbf{a}}$. We therefore construct the set $\bar{\mathbf{A}}_e$ which contains the complement of every explored point. We use this set instead of $\mathbf{A}_u$ to compute our *approximate coverage*:

$$approxcoverage = \frac{1}{n}[n - \max_{\mathbf{a}_u \in \bar{\mathbf{A}}_e} \min_{\mathbf{a}_e \in \mathbf{A}_e} H(\mathbf{a}_e, \mathbf{a}_u)]$$

Computing this measure is only quadratic in $|\mathbf{A}_e|$. The idea is that if we are only going to consider a small set of unexplored points, we will at least consider points known to be at a large (in fact, maximum) distance from at least one explored point. While far from at least one explored point, there is no guarantee that it will not be close to some other explored point. There may also be other unvisited points farther from all explored points than any in $\bar{\mathbf{A}}_e$. Thus, our approximate measure gives a lower bound on the largest gap, which corresponds to an upper bound on coverage.

How well the approximation agrees with the exact version has not been evaluated. For our purposes, we seek practical metrics which give some predictive value. Therefore, we have confined our studies to how predictive the approximate metric is, without worrying about the properties of a metric we can't use in general. How predictive the "true" coverage is remains an interesting research question insofar as one might try to construct other approximations which may offer better predictive value than ours.

With coverage, we run into a problem similar to that encountered with depth. A successful solver may require only a short exploration to solve a problem whereas a poor solver may explore many points during its search. Clearly the relative coverage of two solvers which have sampled differing numbers of points is meaningless.

One way to deal with this would be to consider only a small number of explored points for each solver, but this introduces another parameter or forces us to discard valuable information. Instead, we measure the *rate of coverage*, the rate at which the maximum gap diminishes. This is simply the (approximate) coverage at the end of the search, divided by the number of search steps:

$$rate\,of\,approximate\,coverage = \frac{1}{nT}[n - \max_{\mathbf{a}_u \in \bar{\mathbf{A}}_e} \min_{\mathbf{a}_e \in \mathbf{A}_e} H(\mathbf{a}_e, \mathbf{a}_u)]$$

3 Dual Space Metrics

Of the three metrics we described in detail above, *mobility* and *coverage* are primal space metrics, measuring properties of the set of assignments to the variables. *Depth* is clearly a dual space metric, albeit a very crude summary of the patterns of constraint satisfaction. If we consider a single step in the search, we can view the set of constraints as forming a vector of booleans, much like the variables assignments. It is natural to consider Hamming distances between these vectors. This allows us to adapt the other two metrics to the dual space, giving us *dual mobility* and *dual coverage*. We rename the original two, *primal mobility* and *primal coverage*. The adaptation is fairly straightforward, so we will only describe it briefly.

Every recent successful local search method for satisfiability flips only a single variable at a time. This means the Hamming distance between the assignments in two consecutive steps is at most one. Clearly, this is not the case for the dual space. Changing a single variable can "flip" several constraints. Since we are measuring the "per step" behaviour of algorithms, it is slightly more complicated to track the required information.

However, in the vast majority of SAT instances, typically only a very small fraction of constraints are flipped in a single step, so while tracking something like dual mobility has a worst-case update time that is linear in the number of constraints, in practice the update is more or less constant, and quite small.

3.1 Dual Mobility

The dual mobility metric is almost identical to the primal, but uses vectors of constraint states $\mathbf{c}^{(t)}$ instead of variable assignments. Here, the maximum Hamming distance for a single step is the number of constraints, m. While not as natural a choice as in the primal case, we nonetheless set w to equal m in all of our experiments.

$$dualMobility(w) = \frac{1}{T-w} \sum_{t=1}^{T-w} \frac{H(\mathbf{c}^{(t)}, \mathbf{c}^{(t+w)})}{min(w,m)}$$

The intuitive interpretation is similar as well. If the search persistently revisits a small set of constraint states, it is likely to be trapped. In fact, it may have considerable mobility in variable space but still be oscillating between a small number of constraint states, since there may be many variables that can be flipped without affecting the constraint state (e.g. they could be satisfying or unsatisfying literals in a clause that always has one other literal satisfied).

3.2 Dual Coverage

Again, this new metric is virtually identical to the primal version, substituting the sets of visited constraint configurations, $\mathbf{C}_e$, and the set of complements, $\bar{\mathbf{C}}_e$.

$$rate\,of\,approximate\,dual\,coverage = \frac{1}{mT}[m - \max_{\mathbf{c}_u \in \bar{\mathbf{C}}_e} \min_{\mathbf{c}_e \in \mathbf{C}_e} H(\mathbf{c}_e, \mathbf{c}_u)]$$

Effective local search solvers typically have only a small number of unsatisfied constraints at any given time (i.e. good *depth*). However, the search frequently explores many distinct sets before discovering a solution. While any reasonable solver is unlikely to visit states where many constraints are unsatisfied (e.g. gaps will be predominantly large), it seems likely that coverage of the small sets of violations will be a good indicator of success. Therefore, we expect the dual coverage rates to be small but still informative.

4 Experimental Results

While we are interested in applying these metrics to general CSPs, we have confined our initial studies to SAT solvers, where the primal and dual metrics can be directly compared. We use a selection of standard algorithms, reimplemented by us and instrumented to record the required information. Specifically, we consider GSAT [14], HSAT

[5], WalkSAT [13], Novelty+ [10, 6], and ESGint [12].[2] We use a selection of satisfiable instances from the standard DIMACS and SATLIB [7] collections.

Each problem instance was run 100 times and the results averaged over all runs.[3] Where a problem set contains multiple instances, results were averaged over all runs of all instances. The solvers were allowed 500,000 steps and no restarts. All solvers were run with default parameters.[4]

We report five metrics: average depth (Depth - lower is better), average primal mobility (PMob - higher is better), average primal coverage rate (PCov - higher is better), average dual mobility (DMob - higher is better), and average dual coverage rate (DCov - higher is better). We also report the percentage of runs that failed to find a satisfying solution (Fail% - lower is better), the average steps before terminating (avgSteps - lower is better), and the estimated expected number of steps under an optimal restart scheme (OEESS - lower is better) (see [11] for a full explanation of this measure; in brief, it is an estimate of how many steps would be taken if restarts were used and the restart period tuned).[5] Since the objective here is not to compare the solvers, per se, but rather to understand the relationship between the various metrics and solver performance, this methodology is quite appropriate. For brevity, and because the instrumented solvers incur extra runtime penalties, we only report search steps. The results are very similar when performance is evaluated in terms of runtime. Results are given in Tables 1-10.

Previous experiments [11] showed that high depth implies poor performance but that low depth is relatively uninformative (could indicate a trapped solver). Primal coverage is a much better indicator than depth, and primal mobility is best of all. The study likewise showed that no one of these metrics is sufficient to explain performance but that good rating in all three is necessary. Less conclusively, experiments suggested that the three might represent a set of sufficient conditions.

Turning now to our new results, one uniform observation is that if the failure rate is high, then typically depth is quite small, and mobility and coverage are likewise small. This indicates that the solver spent most of its time trapped in some local minimum. GSAT falls prey to this often, and HSAT, which employs a Tabu-like strategy of preferring the least recently flipped variable when breaking ties, suffers from it only slightly less.

Generally speaking, ranking the methods by the primal and the dual versions of a given metric give the same results. Similarly, high values of primal or dual mobility

[2] ESGint is a pure integer implementation of ESG (previous versions used floating-point for weights). It has several new features, including a non-unit clause list, and decouples the weight increase and smoothing operations as first implemented in SAPS [9]. It behaves almost identically to the original ESG but runs faster and with no numerical drift due to floating point error.

[3] Except for primal and dual coverage results. Coverage is expensive to compute so it was only averaged over the first 10,000 steps in each of 20 runs.

[4] WalkSAT (noise=0.5), Novelty+ (noise=0.5, secondBest=0.01), ESGint (noise=0.01, smoothRate=0.8, smoothProb=0.05, reweightSize=1.3).

[5] There are some very low estimated expected steps in the results for solvers which have performed very poorly. The estimates are very unreliable when failure rates are high. We distinguish these cases with the following symbol:†.

Table 1. Problem set "ais06" : 61v, 518c, All-Interval Series (SATLIB)

Solver	Fail%	Av Steps	OEESS	Depth	PMob	DMob	PCov	DCov
GSAT	94	470002	829†	0.0022	0.0502	0.014	0.00041	0.00027
HSAT	98	490000	1499†	0.0023	0.0504	0.009	0.00078	0.00049
WalkSAT	0	1072	997	0.0050	0.2235	0.156	0.00062	0.00023
Novelty+	0	1932	1761	0.0047	0.1899	0.110	0.00058	0.00024
ESGint	0	595	595	0.0034	0.2282	0.173	0.00219	0.00096

Table 2. Problem set "ais08" : 113v, 1520c, All-Interval Series (SATLIB)

Solver	Fail%	Av Steps	OEESS	Depth	PMob	DMob	PCov	DCov
GSAT	100	500000	500000	0.001178	0.025	0.0017	2.05e-05	9.6e-06
HSAT	100	500000	500000	0.001181	0.026	0.0012	2.10e-05	10.0e-06
WalkSAT	0	29362	10800	0.002816	0.170	0.0447	3.66e-05	12.9e-06
Novelty+	0	44385	35131	0.002712	0.149	0.0322	4.95e-05	18.7e-06
ESGint	0	4985	4954	0.001764	0.178	0.1189	18.15e-05	64.7e-06

Table 3. Problem set "anomaly" : 48v, 261c, Blocks World 3 Blocks, 3 Steps (SATLIB - Kautz and Selman)

Solver	Fail%	Av Steps	OEESS	Depth	PMob	DMob	PCov	DCov
GSAT	79	395007	240†	0.011	0.07	0.048	0.0009	0.0004
HSAT	79	395006	200†	0.009	0.06	0.045	0.0014	0.0006
WalkSAT	0	403	249	0.014	0.22	0.168	0.0023	0.0009
Novelty+	0	598	401	0.016	0.20	0.160	0.0024	0.0011
ESGint	0	96	96	0.006	0.32	0.206	0.0049	0.0018

Table 4. Problem set "medium" : 116v, 953c, Blocks World 5 Blocks, 4 Steps (SATLIB - Kautz and Selman)

Solver	Fail%	Av Steps	OEESS	Depth	PMob	DMob	PCov	DCov
GSAT	90	450009	1308†	0.0059	0.04	0.022	0.00015	0.00007
HSAT	88	440007	622†	0.0060	0.05	0.026	0.00037	0.00017
WalkSAT	0	1085	1085	0.0080	0.19	0.181	0.00091	0.00035
Novelty+	0	1362	847	0.0086	0.20	0.174	0.00046	0.00018
ESGint	0	273	271	0.0042	0.31	0.213	0.00163	0.00060

Table 5. Problem set "bw_large.b" : 1087v, 13772c, Blocks World 11 Blocks, 9 Steps (SATLIB - Kautz and Selman)

Solver	Fail%	Av Steps	OEESS	Depth	PMob	DMob	PCov	DCov
GSAT	100	500000	500000	0.001205	0.0095	0.0068	2.42e-05	1.04e-05
HSAT	100	500000	500000	0.001060	0.0100	0.0065	2.48e-05	1.03e-05
WalkSAT	59	386811	386811	0.001519	0.0571	0.0214	2.76e-05	1.13e-05
Novelty+	69	419241	419241	0.001517	0.0610	0.0148	2.75e-05	1.16e-05
ESGint	0	52559	49560	0.002034	0.1318	0.1053	3.64e-05	1.49e-05

Table 6. Problem set "logistics.a" : 828v, 6718c, Logistics, 8 Packages, 11 Steps (SATLIB - Kautz and Selman)

Solver	Fail%	Av Steps	OEESS	Depth	PMob	DMob	PCov	DCov
GSAT	100	500000	500000	0.00292	0.008	0.0034	1.95e-05	8.81e-06
HSAT	100	500000	500000	0.00285	0.010	0.0032	1.99e-05	9.08e-06
WalkSAT	3	133271	132120	0.00134	0.072	0.0225	2.86e-05	9.03e-06
Novelty+	18	253471	253471	0.00130	0.067	0.0148	2.93e-05	9.31e-06
ESGint	0	13213	9068	0.00198	0.157	0.1365	5.10e-05	16.14e-06

Table 7. Problem set "huge" : 459v, 7054c, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman)

Solver	Fail%	Av Steps	OEESS	Depth	PMob	DMob	PCov	DCov
GSAT	99	495003	39500†	0.00210	0.0184	0.007	2.39e-05	1.10e-05
HSAT	100	500000	500000	0.00206	0.0181	0.005	2.41e-05	1.12e-05
WalkSAT	0	21831	20852	0.00251	0.0926	0.127	3.65e-05	1.50e-05
Novelty+	0	19395	19155	0.00261	0.0934	0.130	3.27e-05	1.33e-05
ESGint	0	2971	2971	0.00300	0.2297	0.216	16.42e-05	6.57e-05

Table 8. Problem set "ii08" : 14 instances, Inductive Inference (DIMACS - Resende)

Solver	Fail%	Av Steps	OEESS	Depth	PMob	DMob	PCov	DCov
GSAT	35	186597	120342	0.00138	0.257	0.103	0.0007	0.00032
HSAT	49	243184	148152	0.00144	0.224	0.108	0.0014	0.00044
WalkSAT	0	530	527	0.01212	0.230	0.186	0.0016	0.00095
Novelty+	0	1268	1268	0.00604	0.199	0.213	0.0012	0.00078
ESGint	0	344	316	0.00268	0.180	0.205	0.0014	0.00100

Table 9. Problem set "par08" : 5 instances, Unsimplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford)

Solver	Fail%	Av Steps	OEESS	Depth	PMob	DMob	PCov	DCov
GSAT	96.0	485223	472390	0.0128	0.29008	0.016	8.10e-05	2.42e-05
HSAT	99.4	497004	406017	0.0136	0.36490	0.009	7.85e-05	1.48e-05
WalkSAT	9.8	163143	83621	0.0321	0.13042	0.063	7.28e-05	2.50e-05
Novelty+	0.2	59040	36949	0.0215	0.10570	0.047	14.91e-05	5.20e-05
ESGint	0.0	9429	8774	0.0139	0.13048	0.053	37.07e-05	10.53e-05

Table 10. Problem set "uf250" : 100 instances, 250v, 1065c, hard, random 3-SAT (SATLIB)

Solver	Fail%	Av Steps	OEESS	Depth	PMob	DMob	PCov	DCov
GSAT	90.0	456413	276154	0.0024	0.068	0.004	3.54e-05	0.75e-05
HSAT	88.3	441720	58028	0.0024	0.099	0.013	10.35e-05	1.90e-05
WalkSAT	1.5	40660	34723	0.0075	0.133	0.034	9.54e-05	2.40e-05
Novelty+	3.1	59397	36017	0.0061	0.101	0.031	8.11e-05	2.23e-05
ESGint	0.1	18581	13108	0.0066	0.255	0.054	20.24e-05	4.04e-05

and coverage are good predictors of performance, in terms of average steps, and also in terms of expected steps (when high failure rates do not render the estimate meaningless). This seems to correspond to the earlier study's view that good depth, mobility and coverage are necessary, but it now appears we can consider either the primal or the dual versions.

Only one data set shown here corresponds to random instances (10). The rest are so-called *structured* instances. This provides fresh evidence for the general applicability of the primal metrics since the earlier study focused on random instances.

We note the following specifics:

- Tables 1 and 8 both show the dual metrics giving a better general prediction of performance.
- Table 9 is the only example where the dual version (of mobility in this case) is significantly worse predictive. However, performance appears to be more related to coverage in this instance, and the dual coverage is nicely predictive.
- Table 10 shows HSAT with high primal coverage, but poor performance. The dual coverage much better reflects performance. Mobility is similar.

In summary, these results show a (to us) surprisingly close relationship between primal and dual versions of metrics. They are almost equally predictive. Our expectations were that there would be more disparity due to local searchers becoming trapped in large basins, thus achieving high primal mobility but visting only a small set of constraint states (low dual mobility).

In one sense, this result is disappointing, because it suggests that the dual metrics have revealed little new information about the search. However, we believe it might be because we have not examined enough scenarios where local searchers fail. The only marked failures in our study are GSAT/HSAT, which are trivially stuck in local minima, and therefore easily explained by primal metrics alone. If, for example, the studies were scaled to larger structured instances, where even the best solvers tend to perform poorly, we might be able to explain some of the failures by comparing primal and dual metrics. We intend to perform such studies in the near future. More generally, we believe it is worth examining both the primal and dual metrics whenever a failure in local search is encountered, if only to rule out possibilities.

As far as general CSPs are concerned, the outlook is very positive. The results offer a strong hope that the dual metrics will offer good predictions for the local search CSP solvers. We plan to test this experimentally as well. An interesting corresponding study could examine the metrics on the general CSPs and on SAT solvers solving CNF-encoded versions of the same problems.

Finally, the new results obtained for structured instances show that the predictive value of the original primal metrics are not limited to the random 3-SAT instances (uf250 in this study), but extend across a range of problem classes.

5 Related Metrics Research

Metrics often play a background role in local search research, used behind the scenes to study or tune algorithms. We will now discuss related work with metrics in order to show how much metrics similar to those presented here have been involved in the evolution of local search.

Attempts to characterize local search behaviour crop up frequently in the literature, although they are typically formulated in order to automatically tune search parameters. Cha and Iwama use a notion similar to mobility in analyzing the behaviour of weighted GSAT [2, 3] by explicitly counting how many previously unvisited assignments are reached in a given period of the search. They were working with small instances where such a measure is not unreasonable. They also plot the Hamming distance between assignments and the eventual solution.

In an attempt to tune the ’noise’ parameters for algorithms such as WalkSAT and Novelty, both of which contain a random walk component, McAllester et al. [10] use two metrics that they demonstrate to give similar values across six different algorithms (WalkSAT, Novelty, and some variants) when each algorithm is optimally tuned. That is, there is a value for these metrics which corresponds to optimal performance, regardless of the method. The two metrics are:

- average number of unsatisfied clauses (noise level invariant)
- mean over variance of number of unsatisfied clauses (optimality invariant)

The first closely resembles the depth metric. Its invariant property means that if one knows the average depth for an algorithm when optimally tuned, one could obtain optimal tunings for some other algorithm by trying to obtain the same average depth. The second metric is a more complex view of the behaviour of the objective function that manages to capture some notion of mobility. The intuition is that, if the variance in objective value over the course of the search is low, then it is likely that the search is trapped. The evidence presented suggests that nearly minimizing this ratio will provide an optimal noise tuning. While this research focuses on the issue of how to tune noise parameters in particular, it is similar in spirit to our own work in attempting to characterize behaviour over several different algorithms.

In their work on *reactive search* for MAX-SAT [1], Batitti notes that there is a tradeoff between *bias* (essentially the same as depth) and *diversification* (similar to mobility[6]). They examine how various features found in local search methods for SAT and MAX-SAT tradeoff these two metrics in an effort to understand what features are effective. Additionally, they use their diversity metric to automatically tune the list length parameter of a Tabu search method.

Something like mobility and depth are proposed in [16], along with several other metrics. The goal in this case was to automatically tune the many parameters that control DLM-99 [17]. Among the metrics they considered for this purpose are:

- least number of unsatisfied clauses (LUC)
- average number of unsatisfied clauses (AUC)
- Hamming distances computed over a history (DIS)

[6] The diversification measure used is the average over all explored assignments of the assignments’ Hamming distances from the initial assignment. This is an interesting measure but clearly has some problems since a search trapped at some distance from the initial assignment will appear to have high diversity.

There are several other metrics relating to clause weights and other features of DLM. These metrics are not independent of the solver, so we do not consider them in the same light.[7] Clearly, AUC is just like our average depth measure (although without the initial skipped steps). DIS, although not clearly explained, appears to be similar to mobility, and is motivated in a similar manner by the authors. Trial and error experimentation led the authors to conclude that LUC and DIS were valuable measures (along with two relating to clause weights). They use these measures after a short run to decide which of five parameters sets to use for DLM-99, according to a hand-crafted heuristic. They did not apply these measures to other solvers.

It is interesting that they found LUC more informative than AUC, and it might be worth exploring the predictive value of LUC. The predictive value of DIS agrees well with our own results.

6 Conclusions

The results presented here strongly suggest that the new dual metrics are at least as effective as the original primal versions, and possibly more so. Furthermore, both primal and dual metrics are effective on structured instances. This offers a strong hope that these metrics can be effectively applied to local search solvers for general CSPs and forms a key part of extending this research. Aside from that, more systematically measuring the relationship between the various metrics and solver performance would strengthen our belief in their efficacy. Exploring a even wider range of problems and solvers, particularly in cases where solvers fail, would help establish their generality.

Acknowledgements

This research was supported by the Alberta Ingenuity Fund. Thanks to Rob Holte, Dale Schuurmans, and Toby Walsh for useful discussions.

References

1. Roberto Battiti. Reactive search: Toward self–tuning heuristics. In V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors, *Modern Heuristic Search Methods*, pages 61–83. John Wiley & Sons Ltd., Chichester, 1996.

[7] It is worth noting that some of these DLM-related measures have some relationship to counting the number of times the search reached a local minimum, something we have not examined systematically but which seems particularly interesting to clause weighting methods that adjust weights at local minima (e.g. DLM, SDF, ESG, SAPS, and PAWS). In particular, they consider the ratio of weight updates to search steps (dual steps over primal steps in ESG parlance), a measure which our own informal experimentation has shown to have a clear relationship to optimal parameter settings for ESG/SAPS. However, our own limited investigation has not led to any reliable way to exploit this metric.

2. Byungki Cha and Kazuo Iwama. Performance Test of Local Search Algorithms Using New Types of Random CNF Formulas. In *Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 304–311. Kyushu University, 1995.
3. Byungki Cha and Kazuo Iwama. Adding New Clauses for Faster Local Search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 332–337, 1996.
4. Moti Frances and Ami Litman. On Covering Problems of Codes. In *Theory of Computing Systems*, volume 30, pages 113–119, March 1997.
5. Ian P. Gent and Toby Walsh. Towards an Understanding of Hill-Climbing Procedures for SAT. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-93)*, pages 28–33, 1993.
6. Holger Hoos. On the Run-time Behaviour of Stochastic Local Search Algorithms for SAT. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, pages 661–666. MIT Press, 1999.
7. Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In I. P. Gent, H von Maaren, and T. Walsh, editors, *Proceedings of the 3rd International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2000)*, pages 283–292. IOS Press, 2000.
8. Eric Horvitz, Yongshao Ruan, Carla P. Gomes, Henry Kautz, Bart Selman, and David Maxwell Chickering. A Bayesian approach to tackling hard computational problems. In *Proceedings of the 17th Conference on Uncertainty and Artificial Intelligence (UAI 2001)*, pages 235–244, August 2001.
9. Frank Hutter, Dave A. D Tompkins, and Holger H. Hoos. Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT. In *Proceedings of the 8th International Conference on the Principles and Practice of Constraint Programming (CP-2002)*, 2002.
10. David McAllester, Bart Selman, and Henry Kautz. Evidence for Invariants in Local Search. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*. AT&T Laboratories, 1997.
11. Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, 132(2):121–150, 2001.
12. Dale Schuurmans, Finnegan Southey, and Robert C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In *Proceedings of 17th International Joint Conference on Artificial Intelligence (IJCAI-2001)*, volume 1, pages 334–341, 2001.
13. Bart Selman, Henry A. Kautz, and Bram Cohen. Noise Strategies for Improving Local Search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*. AT&T Bell Labs, July 1994.
14. Bart Selman, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446. AT&T, University of Toronto, Simon Fraser, 1992.
15. J. R. Thornton, D. N. Pham, S. Bain, and V. Ferreira Jr. Additive versus Multiplicative Clause Weighting for SAT. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-2004)*, 2004.
16. Zhe Wu and Benjamin W. Wah. Solving Hard Satisfiability Problems: A Unified Algorithm Based On Discrete Lagrange Multipliers. In *Proceedings of 11th IEEE Conference on Tools with Artificial Intelligence*. UIUC, November 1999.
17. Zhe Wu and Benjamin W. Wah. Trap Escaping Strategies in Discrete Lagrangian Methods for Solving Hard Satisfiability and Maximum Satisfiability Problems. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, 1999.

Input Distance and Lower Bounds for Propositional Resolution Proof Length

Allen Van Gelder

University of California, Santa Cruz CA 95060, USA
http://www.cse.ucsc.edu/~avg

Abstract. Input Distance (Δ) is introduced as a metric for propositional resolution derivations. If $\mathcal{F} = C_i$ is a formula and D is a clause, then $\Delta(D, \mathcal{F})$ is defined as $\min_i |D - C_i|$. The Δ for a derivation is the maximum Δ of any clause in the derivation. Input Distance provides a refinement of the clause-width metric analyzed by Ben-Sasson and Wigderson (JACM 2001) in that it applies to families whose clause width grows, such as pigeon-hole formulas. They showed two upper bounds on $(W - \mathit{width}(\mathcal{F}))$, where W is the maximum clause width of a narrowest refutation of $\mathcal{F}$. It is shown here that (1) both bounds apply with $(W - \mathit{width}(\mathcal{F}))$ replaced by Δ; (2) for pigeon-hole formulas $\mathrm{PHP}(m, n)$, the minimum Δ for any refutation is $\Omega(n)$. A similar result is conjectured for the $GT(n)$ family analyzed by Bonet and Galesi (FOCS 1999).

1 Introduction

The reader is assumed to be generally familiar with the propositional satisfiability problem, CNF formulas, and resolution derivations. Some definitions are briefly reviewed in Section 2, but are not comprehensive.

Ben-Sasson and Wigderson [3] showed that, if the minimum-length general resolution refutation for a CNF formula $\mathcal{F}$ has S steps, and if the minimum-length tree-like refutation of $\mathcal{F}$ has S_T steps, then there is a (possibly different) refutation of $\mathcal{F}$ using clauses of width at most:

$$w(\mathcal{F} \vdash \bot) \leq w(\mathcal{F}) + c\sqrt{n \ln S}; \tag{1}$$

$$w(\mathcal{F} \vdash \bot) \leq w(\mathcal{F}) + \lg S_T. \tag{2}$$

Note that the $w(\mathcal{F})$ terms were omitted from their statement in the introduction, but appear in their statements of the theorems. The notation for this expression is:

- n is the number of propositional variables in $\mathcal{F}$;
- $w(\mathcal{F})$ is the width of the widest clause in $\mathcal{F}$;
- $w(\mathcal{F} \vdash \bot)$ denotes the minimum *resolution width* of π ranging over all resolution derivations that refute $\mathcal{F}$, where the *resolution width* of π, denoted $w_{\mathcal{F}}(\pi)$, is the width of the widest clause in π;
- c is a constant, independent of $\mathcal{F}$;
- ln and lg denote natural and binary logs, respectively.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 282–293, 2005.

All formulas and clauses are propositional, clauses are disjunctions of literals, formulas are in CNF, unless specified otherwise.

Our main results essentially eliminate the $w(\mathcal{F})$ terms in the Ben-Sasson and Wigderson theorems [3], and replace resolution width by $\Delta_{\mathcal{F}}(\pi)$, the *input distance*, as defined next, in Section 1.1. For families of formulas whose widest clause is bounded by a constant, input distance and resolution width are essentially equivalent measures.

1.1 Input Distance

We define *input distance* for nontautologous clauses (primarily derived clauses in a resolution proof) for input CNF formula $\mathcal{F}$.

Definition 1.1. (input distance) All clauses mentioned are non-tautologous. Let D be a clause; let C be an input clause, i.e., a clause of formula $\mathcal{F}$. The *input distance* of D from C is $|D - C|$, treating D and C as sets of literals, and using "$-$" for set difference. The *input distance* of D from $\mathcal{F}$, denoted $\Delta_{\mathcal{F}}(D)$, is the minimum over $C \in \mathcal{F}$ of the *input distances* of D from C.

For a resolution proof π the *input distance* of π from $\mathcal{F}$, denoted $\Delta_{\mathcal{F}}(\pi)$, is the maximum over $D \in \pi$ of the *input distances* of D from $\mathcal{F}$.

When $\mathcal{F}$ is understood from the context, $\Delta(D)$ and $\Delta(\pi)$ are written. Following Ben-Sasson and Wigderson [3], $\Delta(\mathcal{F} \vdash D)$ denotes the minimum of $\Delta_{\mathcal{F}}(\pi)$ over all π that are derivations of D from $\mathcal{F}$.

1.2 Summary of Results

The theorems shown here are that, if π is a resolution refutation of $\mathcal{F}$ and π uses all clauses of $\mathcal{F}$ and the length of π is S, then there is a refutation of $\mathcal{F}$ using clauses that have *input distance* from $\mathcal{F}$ that is at most:

$$\Delta(\mathcal{F} \vdash \bot) \leq c\sqrt{n \ln S}; \tag{3}$$

$$\Delta(\mathcal{F} \vdash \bot) \leq \lg S_T. \tag{4}$$

Also, we show that the pigeon-hole family of formulas $\mathrm{PHP}(m, n)$ require refutations with input distance $\Omega(n)$, although they contain clauses of width n. This result suggests that input distance provides a refinement of the clause-width metric as a measure of resolution difficulty. That is, when a family of formulas with increasing clause-width, such as $\mathrm{PHP}(m, n)$, is transformed into a bounded-width family, such as $\mathrm{EPHP}(m, n)$, and the bounded-width family has large resolution width, this is not simply because they rederive the wide clauses of the original family, then proceed to refute the original family. Rather, it is the case that wide clauses substantially different from those in the original family must be derived. However, note that input distance $\Omega(n)$ does not imply any useful lower bound on general resolution refutation length for $\mathrm{PHP}(m, n)$, at least not through any known theorem.

2 Preliminaries

2.1 Notation

This section collects notations and definitions used throughout the paper. Standard terminology for conjunctive normal form (CNF) formulas is used. Notations are summarized in Table 1. Although the general ideas of resolution and derivations are well known, there is no standard notation for many of the technical aspects, so it is necessary to specify our notation in detail.

Table 1. Summary of notations

$a, \ldots, z$	Literal; i.e., propositional variable or negated propositional variable.
$\neg x$	Complement of literal x; $\neg\neg x$ is not distinguished from x.
$\lvert x \rvert$	The propositional variable in literal x; i.e., $\lvert a \rvert = \lvert \neg a \rvert = a$.
$A, \ldots, Z$	Disjunctive clause, or set of literals, depending on context.
$\mathcal{A}, \ldots, \mathcal{H}$	CNF formula, or set of literals, depending on context.
π	Resolution derivation DAG.
σ	Total assignment, represented as the set of true literals.
$[p_1, \ldots, p_k]$	Clause consisting of literals $p_1, \ldots, p_k$.
$\bot$	The *empty clause*, which represents *false*.
$\top$	The *tautologous clause*, which represents *true*; (see Definition 2.2).
$\alpha, \ldots, \delta$	Subclause, in the notation $[p, q, \alpha]$, denoting a clause with literals p, q, and possibly other literals, α.
C^-	Read as "C, or some clause that subsumes C".
p	In a context where a unit clause is expected, $[p]$ may be abbreviated to p.
C, p	In a context where a formula is expected, $\{C\}$ may be abbreviated to C and $\{[p]\}$ may be abbreviated to p.
$+$, $-$	Set union and difference, as infix operators, where operands are formulas, possibly using the abbreviations above.
$\mathbf{res}(q, C, D)$	Resolvent of C and D, where q and $\neg q$ are the clashing literals (see Definition 2.2).
$C \vert \mathcal{A}$, $\mathcal{F} \vert \mathcal{A}$, $\pi \vert \mathcal{A}$	C (respectively $\mathcal{F}$, π) *strengthened* by $\mathcal{A}$ (see Definition 2.4).

Definition 2.1. (assignment, satisfaction, model) A partial assignment is a partial function from the set of variables into $\{\mathit{false}, \mathit{true}\}$. This partial function is extended to literals, clauses, and formulas in the standard way. If the partial assignment is a total function, it is called a *total assignment*, or simply an *assignment*.

A clause or formula is *satisfied* by a partial assignment if it is mapped to *true*; A partial assignment that satisfies a formula is called a *model* of that formula.

□

A partial assignment is conventionally represented by the (necessarily consistent) set of *unit clauses* that are mapped into *true* by the partial assignment. Note that this representation is a very simple formula.

2.2 Resolution as a Total Function

Definition 2.2. (resolution, subsumption, tautologous) A clause is *tautologous* if it contains complementary literals. All tautologous clauses are considered to be indistinguishable and are denoted by $\top$.

If $C = [q, \alpha]$ and $D = [\neg q, \beta]$ are two non-tautologous clauses (α and β are subclauses), then

$$\mathbf{res}(q, C, D) = \mathbf{res}(q, D, C) = \mathbf{res}(\neg q, C, D) = \mathbf{res}(\neg q, D, C) = [\alpha, \beta]$$

defines the *resolution* operation, and $[\alpha, \beta]$ is called the *resolvent*, which may be tautologous. Resolution is extended to include $\top$ as an identity element:

$$\mathbf{res}(q, C, \top) = C$$

provided C contains q or $\neg q$.

Resolution is further extended to apply any two non-tautologous clauses and any literals, as follows. Fix a total order on the clauses definable with the n propositional variables such that $\bot$ is smallest, $\top$ is largest, and wider clauses are "bigger" than narrower clauses. Other details of the total order are not important.

If $C = [\alpha]$ does not contain q and $D = [\neg q, \beta]$ is non-tautologous, then

$$\mathbf{res}(q, C, D) = \mathbf{res}(q, D, C) = \mathbf{res}(\neg q, C, D) = \mathbf{res}(\neg q, D, C) = [\alpha]$$

If $C = [\alpha]$ and $D = [\beta]$ and neither contains q or $\neg q$, and both are non-tautologous, then

$$\begin{aligned}\mathbf{res}(q, C, D) = \mathbf{res}(q, D, C) = \mathbf{res}(\neg q, C, D) = \mathbf{res}(\neg q, D, C)\\ = \text{the smaller of } C \text{ and } D.\end{aligned}$$

With this generalized definition of resolution, we have an algebra, and the set of clauses (including $\top$) is a lattice, based on $\subseteq$, with the convention that every clause is a subset of $\top$. We shall see later that the benefit of this structure is that resolution "commutes up to subsumption" with *strengthening* (see Definition 2.4), so strengthening can be applied to any resolution derivation to produce another derivation.

If clause $C \subset D$, we say C *properly subsumes* D; if $C \subseteq D$, we say C *subsumes* D. Also, any non-tautologous clause properly subsumes $\top$. Notation D^- is read as "D, or some clause that subsumes D". □

Definition 2.3. (derivation, refutation) A *derivation* (short for *propositional resolution derivation*) from formula $\mathcal{F}$ is a *rooted*, directed acyclic graph (DAG) in which each vertex is labeled with a clause and possibly with a clashing literal. Let D be the clause label of vertex v. If $D = C \in \mathcal{F}$, then v has no out-edges and no clashing literal, and is called a *leaf*. Otherwise v is called a *resolution* *vertex*, has two out-edges, say to vertices with clause labels D_1 and D_2, and is also labeled with the clashing literal q such that

$$D = \mathbf{res}(q, D_1, D_2),$$

where **res** is the total function defined in Definition 2.2. In much of the discussion, vertices are referred to by their clause labels.

A derivation derives its root clause. When the root clause is $\bot$, the derivation is called a *refutation.* □

2.3 The Strengthening Operation

Definition 2.4. (strengthened formula, strengthened derivation) Let $\mathcal{A}$ be a partial assignment for formula $\mathcal{F}$. Let π be a derivation from $\mathcal{F}$. The clause $C|\mathcal{A}$, read "C strengthened by $\mathcal{A}$", and the formula $\mathcal{F}|\mathcal{A}$, read "$\mathcal{F}$ strengthened by $\mathcal{A}$", are defined as follows.

1. $C|\mathcal{A} = \top$, if C contains any literal that occurs in $\mathcal{A}$.
2. $C|\mathcal{A} = C - \{q \mid q \in C \text{ and } \neg q \in \mathcal{A}\}$, if C does not contain any literal that occurs in $\mathcal{A}$. This may be the empty clause.
3. $\mathcal{F}|\mathcal{A} = \big\{C|\mathcal{A} \bigm| C \in \mathcal{F}\big\}$; i.e., apply strengthening to each clause in $\mathcal{F}$. Usually, occurrences of $\top$ (produced by part (1)) are deleted in $\mathcal{F}|\mathcal{A}$.
4. $\pi|\mathcal{A}$ is the same DAG as π structurally, but the clauses labeling the vertices are changed as follows. If a leaf (input clause) of π contains C, then the corresponding leaf of $\pi|\mathcal{A}$ contains $C|\mathcal{A}$. Each derived clause of $\pi|\mathcal{A}$ uses resolution on the same clashing literal as the corresponding vertex of π.

The operation $\mathcal{F}|p$ (i.e., $\mathcal{F}|\{[p]\}$) is sometimes called "unit simplification". □

The term "strengthen" comes from the theorem-proving community [7]. Ben-Sasson and Wigderson [3] and others in the proof-complexity community use the term "restriction" for "unit simplification" or "strengthening by a single literal"; several different terms for this operation may be found in the literature.

Example 2.5. Let $\mathcal{F}$ consist of clauses $C_1 = [a, b]$, $C_2 = [\neg a, c]$, $C_3 = [\neg b, e]$, and $C_4 = [\neg c, \neg d]$. Let π consist of leaves C_1, C_2 and C_4 and the derived clauses

$$\begin{aligned} D_1 &= \mathbf{res}(a, C_1, C_2) = [b, c]\,, \\ D_2 &= \mathbf{res}(c, D_1, C_4) = [b, \neg d]\,, \\ D_3 &= \mathbf{res}(b, D_2, C_3) = [e, \neg d]\,. \end{aligned}$$

Then $\mathcal{F}|a = \{[c]\,, [\neg b, e]\,, [\neg c, \neg d]\}$, Also, $\mathcal{F}|\{a, c\} = \{[\neg b, e]\,, [\neg d]\}$.

Now consider $\pi|a$. The leaves are $C_1|a = \top$, $C_2|a = [c]$, $C_3|a = C_3$, and $C_4|a = C_4$. The derived clauses are E_1, E_2 and E_3, where:

$$\begin{aligned} E_1 &= \mathbf{res}(a, \top, [c]) = [c]\,; \\ E_2 &= \mathbf{res}(c, [c]\,, [\neg c, \neg d]) = [\neg d]\,; \\ E_3 &= \mathbf{res}(b, [\neg d]\,, [\neg b, e]) = [\neg d]\,. \end{aligned}$$

Notice that $E_i \neq D_i|a$ in any case, but $E_i = (D_i|a)^-$ in all cases. Also notice that the clashing literal is absent from one operand in the resolution for E_3, so the resolvent is just the other operand. □

Lemma 2.6. Given formula $\mathcal{F}$, and a strengthening literal p,

$$\mathbf{res}(q, D_1|p, D_2|p) \subseteq \mathbf{res}(q, D_1, D_2)|p.$$

Proof. The principal case that requires checking is when $q = p$ and $q \in D_1$ and $\neg q \in D_2$ (or *vice versa*). In this case,

$$\mathbf{res}(q, D_1, D_2)|p = \mathbf{res}(q, D_1, D_2) = (D_1 - p) \cup (D_2 - \neg p).$$

Then $\mathbf{res}(q, D_1|p, D_2|p) = D_2|p = (D_2 - \neg p)$. Therefore, $D_2|p \subseteq \mathbf{res}(q, D_1, D_2)|p$. □

Lemma 2.7. Given formula $\mathcal{F}$, and a strengthening literal p, if π is a derivation of C from $\mathcal{F}$, then $\pi|p$ is a derivation of $(C|p)^-$ (a clause that subsumes $C|p$) from $\mathcal{F}|p$.

Proof. The proof is by induction on the structure of π with edge $v \rightarrow w$ interpreted to mean that v is greater than w. Thus, the base cases are the vertices that are clauses in $\mathcal{F}$, called the leaves. By Lemma 2.6, if a vertex of π contains the derived clause C, and the two adjacent operand vertices satisfy the lemma, then the corresponding vertex of $\pi|p$ contains $(C|p)^-$. □

If C is the root of π and $C|p \neq \top$, then a $\top$-free derivation of $(C|p)^-$ can be constructed from $\pi|p$ by changing all resolution vertices that have exactly one $\top$ operand to "copy" vertices that use the non-$\top$ operand, then deleting all the $\top$ vertices, then compressing out all the copy vertices. Finally, the resulting DAG might have multiple sources, so delete all vertices that cannot be reached from the original root, which now contains $(C|p)^-$. This procedure does not change the *clause* in any vertex of $\pi|p$.

Notice that Ben-Sasson and Wigderson [3] define $\pi|p$ differently, as clause-by-clause strengthening (restriction, in their terminology) of the originally derived clauses. As Example 2.5 showed, this definition does not necessarily produce a derivation; they do not discuss this issue. The definitions used herein *do* ensure that the strengthening of a derivation is a derivation, without using weakening. The point of Lemma 2.7 is that the clauses derived from the strengthened formula are at least as strong as the clause-by-clause strengthenings of the originally derived clauses.

2.4 Input Distance and Strengthening

A few properties of input distance on clauses that result from strengthening are stated.

Lemma 2.8. Let C be a clause of $\mathcal{F}$ and let $\mathcal{A}$ be a partial assignment. If $C|\mathcal{A} \neq \top$ (i.e., $\mathcal{A}$ does not satisfy C), then $\Delta_{\mathcal{F}}(C|\mathcal{A}) = 0$.

Proof. $|(C|\mathcal{A}) - C| = 0$. □

Lemma 2.9. Let D be a clause of $\mathcal{F}$, let $\mathcal{A}$ be a partial assignment, and let $\mathcal{G} = \mathcal{F}|\mathcal{A}$. If $D|\mathcal{A} \neq \top$ (i.e., $\mathcal{A}$ does not satisfy D), then $\Delta_{\mathcal{F}}(D) \leq \Delta_{\mathcal{G}}(D|\mathcal{A}) + |\mathcal{A}|$.

Proof. Suppose $C|\mathcal{A} \in \mathcal{G}$ is a clause for which $\Delta_{\mathcal{G}}(D|\mathcal{A}) = |(D|\mathcal{A}) - (C|\mathcal{A})|$. Then $|D - C| \leq |D - (C|\mathcal{A})| \leq |(D|\mathcal{A}) - (C|\mathcal{A})| + |\mathcal{A}|$. □

3 Size vs. Input Distance Relationships

Ben-Sasson and Wigderson [3] derived size-width relationships that they describe as a "direct translation of [CEI96] to resolution derivations." Their informal statement, "if $\mathcal{F}$ has a *short* resolution refutation then it has a refutation with a small *width*," applies only when $\mathcal{F}$ has no wide clauses.

This section shows that by using input distance rather than clause width, the restriction on the width of $\mathcal{F}$ can be removed. That is, the relationships are strengthened by removing the additive term, *width*$(\mathcal{F})$.

The use of strengthening for recursive construction of refutations with special properties originates with Anderson and Bledsoe [2], who used it as a uniform framework for showing completeness of various restrictions on resolution, including linear resolution, set-of-support strategy, positive resolution, and others. Clegg *et al.* [5] used it in connection with Groebner-basis refutations. Ben-Sasson and Wigderson [3] used it to construct resolution refutations of small width. We use it here to construct resolution refutations of small input distance, closely following Ben-Sasson and Wigderson.

Lemma 3.1. Given formula $\mathcal{F}$, and a strengthening literal p, let $\mathcal{G} = \mathcal{F}|p$. If derivation π_1 derives clause D from $\mathcal{G}$ with input distance $\Delta_{\mathcal{G}}(\pi_1) = (d-1)$, then there is a derivation π_2 that derives $(D + \neg p)^{-}$ from $\mathcal{F}$ with input distance $\Delta_{\mathcal{F}}(\pi_2) \leq d$.

Proof. Since $\mathcal{G}$ contains neither p nor $\neg p$, we can assume w.l.o.g. that no vertices of π_1 have p or $\neg p$ as the clashing literal. Define π_2 to have the same DAG structure as π_1, and the same clashing literal at each vertex, but wherever a leaf of π_1 is labeled with $C|p$, label the corresponding leaf of π_2 with C. Each clause of $\mathcal{F}$ has at most one additional literal, $\neg p$, compared to the corresponding clause of $\mathcal{G}$, or else contains p. But no clauses of $\mathcal{F}$ containing p are leaves of π_2. Complete the clause labeling of π_2 according to the definition of resolution. Clearly π_2 derives $(D + \neg p)^{-}$. For each clause E in π_2, the corresponding clause in π_1 is $E|p$. By Lemma 2.9, $\Delta_{\mathcal{F}}(E) \leq \Delta_{\mathcal{G}}(E|p) + 1$. So $\Delta_{\mathcal{F}}(\pi_2) \leq d$. □

Lemma 3.2. Given formula $\mathcal{F}$, and a strengthening literal p, let $\mathcal{G} = \mathcal{F}|p$ and $\mathcal{H} = \mathcal{F}|\neg p$. If derivation π_1 derives $\bot$ from $\mathcal{G}$ with input distance $\Delta_{\mathcal{G}}(\pi_1) = d-1$, and derivation π_2 derives $\bot$ from $\mathcal{H}$ with input distance $\Delta_{\mathcal{H}}(\pi_2) = d$, then there is a derivation π_3 that derives $\bot$ from $\mathcal{F}$ with input distance $\Delta_{\mathcal{F}}(\pi_3) \leq d$.

Proof. Using Lemma 3.1, there is a derivation π_4 that derives $[\neg p]^-$ from $\mathcal{G}$ with input distance $\Delta_{\mathcal{F}}(\pi_4) \leq d$. If the root of π_4 is $\bot$, let $\pi_3 = \pi_4$ and we are done. Otherwise, construct π_3 as follows:

1. Use π_4 as the initial part of π_3. This part of π_3 has input distance at most d from $\mathcal{F}$.
2. Resolve every clause of $\mathcal{F}$ that contains p with the root of π_4, which contains $[\neg p]$. Call this set of resolvents $\mathcal{F}_1$. All of these resolvents have input distance 0 from $\mathcal{F}$ (Lemma 2.8), so they do not contribute to $\Delta_{\mathcal{F}}(\pi_3)$; also, they and are in $\mathcal{H}$.
3. Let $\mathcal{F}_2$ consist of those clauses in $\mathcal{F}$ that contain neither $\neg p$ nor p. Note that $\mathcal{F}_1 + \mathcal{F}_2 = \mathcal{H}$.
4. Complete the derivation π_3 according to the derivation π_2, using clauses from $\mathcal{F}_1$ and $\mathcal{F}_2$ in place of $\mathcal{H}$ at the leaves of π_2. Since $|D - C| \leq |(D - C|\neg p)|$ for any clauses, C, D, this part of π_3 has input distance at most d from $\mathcal{F}$.

Thus $\Delta_{\mathcal{F}}(\pi_3) \leq d$. □

Theorem 3.3. Let $\mathcal{F}$ be an unsatisfiable formula on $n \geq 1$ variables and let $d \geq 0$ be an integer. Let S_T be the size of the shortest tree-like refutation of $\mathcal{F}$. If $S_T \leq 2^d$, then $\mathcal{F}$ has a refutation π with input distance $\Delta_{\mathcal{F}}(\pi) \leq d$.

Proof. The proof is by induction on the pair (n, d) with the component-wise partial order, and follows Ben-Sasson and Wigderson [3], except that it uses input distance and Lemma 3.2 above. The bases cases are $d = 0$ or $n = 1$, and are immediate. For $d > 0$ and $n > 1$ assume the theorem holds for smaller pairs. Let x be the clashing literal at the root of π, a shortest tree-like refutation of $\mathcal{F}$. The children of the root are themselves the roots of tree-like derivations of x and $\neg x$; call them π_1 and π_0. Assume the size of π_1 is at most 2^{d-1}. But $\pi_1|\neg x$ is a tree-like refutation of $\mathcal{G} = \mathcal{F}|\neg x$. By the inductive hypothesis, $\mathcal{G}$ has a refutation π_2 with input distance $\Delta_{\mathcal{G}}(\pi_2) \leq d - 1$. Also, $\mathcal{H} = \mathcal{F}|x$ has at most $n - 1$ variables, so by the inductive hypothesis, $\mathcal{H}$ has a refutation with input distance $\Delta_{\mathcal{H}}(\pi_1) \leq d$. By Lemma 3.2, $\mathcal{F}$ has a refutation π with input distance $\Delta_{\mathcal{F}}(\pi) \leq d$. □

Corollary 3.4. $S_T(\mathcal{F}) \geq 2^{\Delta(\mathcal{F} \vdash \bot)}$.

Theorem 3.5. Let $\mathcal{F}$ be an unsatisfiable formula on $n \geq 1$ variables and let $d \geq 0$ be an integer. Let $S(\mathcal{F})$ be the size of the shortest refutation of $\mathcal{F}$. If $S(\mathcal{F}) \leq e^{\frac{d^2}{8n}}$, then $\mathcal{F}$ has a refutation π_1 with $\Delta_{\mathcal{F}}(\pi_1) \leq d$.

Proof. The proof is by induction on the pair (n, d) with the component-wise partial order, and follows Ben-Sasson and Wigderson [3], except that it uses input distance and Lemma 3.2 above. Their local variable d is renamed to f here and denotes the input distance that causes a clause to be classified as *fat*; $f = \lceil \sqrt{2n \ln S(\mathcal{F})} \rceil$. For any derivation π, let π^* be the set of clauses $D \in \pi$ with $\Delta_{\mathcal{F}}(D) > f$. Define $a = 2n/(2n - f)$. The theorem follows from this claim:

Claim: For all $b \geq 0$ and $1 \leq m \leq n$, if formula $\mathcal{G}$ has m variables and π is a refutation of $\mathcal{G}$ and $|\pi^*| < a^b$, then $\Delta(\mathcal{G} \vdash \bot) \leq f + b$.

Setting $b = f$ and $\mathcal{G} = \mathcal{F}$, and using the identity $-\ln(1 - f/2n) > f/2n$, ensures that $a^b \geq S(\mathcal{F})$, so ensures the hypothesis, $|\pi^*| < a^b$, is true. Setting $d = 2f$ proves the theorem.

The claim is proved by induction on on the pair (m, b) with the component-wise partial order. The base cases are $b = 0$ or $m = 1$, for which the claim is immediate, as $|\pi^*| = 0$. For $b > 0$ and $m > 1$, there is some literal x that appears in at least $|\pi^*| f/2n$ clauses of π^*. Let $\pi|x$ be as defined in Definition 2.4. By Lemma 2.7 and the discussion following it, there is a $\top$-free derivation π_1 with the same nontautologous clauses as $\pi|x$. Then $|\pi_1^*| \leq (1 - f/2n)|\pi^*| \leq a^{b-1}$. But π_1 refutes $\mathcal{G}|x$, so by the inductive hypothesis, $\Delta(\mathcal{G}|x \vdash \bot) \leq f + b - 1$. Let π_0 be the $\top$-free version of $\pi|\neg x$, which refutes $\mathcal{G}|\neg x$. Since $\mathcal{G}|\neg x$ has fewer than m variables and $|\pi_0^*| \leq a^b$, by the inductive hypothesis, $\Delta(\mathcal{G}|\neg x \vdash \bot) \leq f + b$. Applying Lemma 3.2 proves the claim. □

Corollary 3.6. $S(\mathcal{F}) \geq e^{\frac{\Delta(\mathcal{F} \vdash \bot)^2}{8n}}$.

4 Pigeon-Hole Formulas

The well-known family of Pigeon-Hole formulas for m pigeons and n holes (PHP(m, n)) is defined by these clauses:

$$\begin{aligned} C_i &= [x_{i,1}, \ldots, x_{i,n}] && \text{for } 1 \leq i \leq m \\ B_{ijk} &= [\neg x_{i,k}, \neg x_{j,k}] && \text{for } 1 \leq i \leq m,\ 1 \leq j \leq m,\ 1 \leq k \leq n. \end{aligned}$$

For the standard version, $m = n + 1$. We shall show that any refutation of PHP(m, n) with $m > n$ has input distance $\Omega(n)$. An (already known) exponential lower bound for tree-like refutations follows by Corollary 3.4, but no useful lower bound for general refutations follows by Corollary 3.6, since the "n" in that corollary is the number of variables, which is nm in the notation of this section. The method follows Ben-Sasson and Wigderson [3], except that it uses input distance and the original PHP clauses of width n.

Theorem 4.1. Any refutation of PHP(m, n) with $m > n$ has input distance at least $n/3 - 2$.

Proof. For $1 \le i \le m$, define

$$\mathcal{A}_i = \{C_i, B_{ijk},\ 1 \le j \le m, 1 \le k \le n\}$$

which consists of all the constraints on pigeon i. Define $\mu(D)$, the *complexity* of a clause D, as the minimum number of $\mathcal{A}_i$'s needed to logically imply D. Then $\mu(\bot) = n + 1$ and $\mu(C) = 1$ where C is any input clause. Suppose I is the index set for a minimum-cardinality set of $\mathcal{A}_i$'s that imply D and $n/3 \le |I| < 2n/3$. That is,

$$\Big(\bigwedge_{i \in I} \mathcal{A}_i\Big) \rightarrow D \tag{5}$$

is a tautology. Such an I must exist, because $\mu(\mathbf{res}(q, D_1, D_2)) \le \mu(D_1) + \mu(D_2)$.

Equation (5) holds if and only if the following is unsatisfiable (note that $\neg(D)$ constitutes a set of unit clauses):

$$\Big(\bigwedge_{i \in I} \mathcal{A}_i\Big) \wedge \neg(D) \tag{6}$$

Let P_0 be the set of pigeons (first index of variables) that have negative literals in D; let P_1 be the set of pigeons that have positive literals in D. If D has at least $n/3$ negative literals, then its input distance is at least $n/3 - 2$; assume this is not the case. Therefore, $I - P_0$ is nonempty.

The plan of the proof is to show that, if $I - P_0$ is nonempty and D has fewer than $n/3$ negative literals, either there is an assignment that satisfies (6) or P_1 has at least $n/3 - 1$ pigeons. Table 2 illustrates some of the notation.

Since $I - P_0$ is nonempty, let $p \in I - P_0$ and let $I^* = I - \{p\}$. Thus p is some pigeon whose hole is not forced by $\neg(D)$. By the minimality of I there is an assignment σ that makes $\neg(D)$ true, makes $\mathcal{A}_p$ false and satisfies $\mathcal{A}_i$ for $i \in I^*$. W.l.o.g. let σ be chosen to have as few positive literals as possible. Then σ sets all $x_{ik} = 0$ for i not in $I \cup P_0 \cup P_1$ and $1 \le k \le n$. Further, σ sets all $x_{pk} = 0$, $1 \le k \le n$, since none of these positive literals occur in $\neg(D)$. Choose a function $k(i)$ for $i \in I^*$ such that $x_{i,k(i)} = 1$ in σ. Necessarily, $k(i_1) \ne k(i_2)$ for distinct $i_1, i_2 \in I^*$. Let K be the set of indexes in the range 1 through n that are *not* in the range of $k(i)$; $|K| \ge n/3 + 2$. These are the holes that are available for pigeon p.

Recall that σ sets $x_{pk} = 0$ for all $k \in K$. Also, σ sets $x_{ik} = 1$ for $i \in I^*$ and $k \ne k(i)$ only if $x_{ik} \in \neg(D)$. If, for any $k \in K$, x_{pk} can be flipped to 1 and x_{ik} can be set to 0 for all $i \ne p$ without falsifying $\neg(D)$, that would create a satisfying assignment for (6). Therefore, for each $k \in K$, $\neg(D)$ contains $\neg x_{pk}$ or x_{ik} for some $i \ne p$. Since $|K| > n/3$ and we assumed D has fewer than $n/3$ negative literals, it must be the case that $\neg(D)$ contains $\neg x_{pk}$ for some $k \in K$.

Finally, we argue that since $\neg(D)$ contains $\neg x_{pk}$, for some $k \in K$, it must contain $\neg x_{ik}$ for all $i \in I^*$. Suppose this fails for some i. Then modify σ by setting $x_{p,k(i)} = 1$, $x_{p,k} = 0$, $x_{i,k(i)} = 0$, and $x_{i,k} = 1$ (see Table 2). This produces a satisfying assignment for (6).

Table 2. Changing σ to expose a faulty index set I, in proof of theorem

$D = [\neg x_{11}, x_{32}, x_{52}]$, $\neg(D) = [x_{11}] \wedge [\neg x_{32}] \wedge [\neg x_{52}]$, $I = \{2, 3, 5\}$, $I^* = \{2, 3\}$.

Original σ

pige–	*holes*			
ons	1	2	3	4
1	1	0	0	0
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0
$p = 5$	0	0	0	0
6	0	0	0	0

i	$k(i)$
2	3
3	4

$K = \{1, 2\}$

Modified σ

pige–	*holes*			
ons	1	2	3	4
1	1	0	0	0
2	0	***1***	***0***	0
3	0	0	0	1
4	0	0	0	0
$p = 5$	0	***0***	***1***	0
6	0	0	0	0

To summarize, if $\neg(D)$ contains $\neg x_{pk}$ for some $k \in K$, then D contains positive literals for at least $n/3 - 1$ different pigeons, i.e.,$|P_1| \geq n/3 - 1$, giving D an input distance of at least $n/3 - 2$. □

5 Conclusion

We proposed the *input distance* metric as a refinement of clause width for studying the complexity of resolution. For families with wide clauses, the trade-off between resolution refutation size and input distance is sharper than the trade-off between resolution refutation size and clause width.

We showed that any refutation of PHP(m, n) requires input distance at least $n/3 - 2$. Moreover, the proof showed that this input distance can arise in two possible ways: by having $n/3$ negative literals in a derived clause, or by having $n/3 - 1$ positive literals *that refer to distinct pigeons.*

We conjecture that a similar exercise can be carried out for the family called GT(n) [6], which has general refutations of polynomial size [8], but for which tree-like refutations are exponential [4]. This family can be modified so that regular refutations are also exponential [1]. Bonet and Galesi introduced a bounded-width variant called MGT(n), and showed that refutations of MGT(n) have width $\Omega(n)$ [4]. However, the complexity function they used does not transfer straightforwardly to a lower bound on input distance, as there are clauses with input distance zero and complexity between $n/3$ and $2n/3$.

Some open problems remain. Can input distance improve the lower bound for *weak* PHP(m, n), where $m >> n$? Ben-Sasson and Wigderson [3] transformed this problem into a family with clause width proportional to $\log m$. Are there other natural families to which input distance can be applied? Is there a trade-off between regular refutation size and input distance?

References

1. Alekhnovich, M., Johannsen, J., Pitassi, T., Urquhart, A.: An exponential separation between regular and unrestricted resolution. In: Proc. 34th ACM Symposium on Theory of Computing. (2002) 448–456

2. Anderson, R., Bledsoe, W.W.: A linear format for resolution with merging and a new technique for establishing completeness. Journal of the ACM **17** (1970) 525–534
3. Ben-Sasson, E., Wigderson, A.: Short proofs are narrow — resolution made simple. JACM **48** (2001) 149–168
4. Bonet, M., Galesi, N.: A study of proof search algorithms for resolution and polynomial calculus. In: Proc. 40th Symposium on Foundations of Computer Science. (1999) 422–432
5. Clegg, M., Edmonds, J., Impagliazzo, R.: Using the Groebner basis algorithm to find proofs of unsatisfiability. In: Proc. 28th ACM Symposium on Theory of Computing. (1996) 174–183
6. Krishnamurthy, B.: Short proofs for tricky formulas. Acta Informatica **22** (1985) 253–274
7. Letz, R., Mayr, K., Goller, C.: Controlled integration of the cut rule into connection tableau calculi. Journal of Automated Reasoning **13** (1994) 297–337
8. Stälmarck, G.: Short resolution proofs for a sequence of tricky formulas. Acta Informatica **33** (1996) 277–280

Sums of Squares, Satisfiability and Maximum Satisfiability

Hans van Maaren and Linda van Norden

Delft University of Technology,
Faculty of Electrical Engineering,
Mathematics and Computer Science,
Department of Software Technology,
Mekelweg 4, 2628 CD Delft, The Netherlands
{H.vanMaaren, L.vanNorden}@ewi.tudelft.nl

Abstract. Recently the Mathematical Programming community showed a renewed interest in Hilbert's Positivstellensatz. The reason for this is that global optimization of polynomials in $\mathbb{R}[x_1, \ldots, x_n]$ is $\mathcal{NP}$-hard, while the question whether a polynomial can be written as a sum of squares has tractable aspects. This is due to the fact that Semidefinite Programming can be used to decide in polynomial time (up to a prescribed precision) whether a polynomial can be rewritten as a sum of squares of linear combinations of monomials coming from a specified set. We investigate this approach in the context of Satisfiability. We examine the potential of the theory for providing tests for unsatisfiability and providing MAXSAT upper bounds. We compare the SOS (Sums Of Squares) approach with existing upper bound and rounding techniques for the MAX-2-SAT case of Goemans and Williamson [8] and Feige and Goemans [6] and the MAX-3-SAT case of Karloff and Zwick [9], which are based on Semidefinite Programming as well. We show that the combination of the existing rounding techniques with the SOS based upper bound techniques yields polynomial time algorithms with a performance guarantee at least as good as the existing ones, but observably better in particular cases.

1 Introduction

Hilbert's Positivstellensatz states that a non-negative polynomial in $\mathbb{R}[x_1, \ldots, x_n]$ is a SOS in case $n = 1$, or has degree $d = 2$, or $n = 2$ and $d = 4$. Despite these restrictive constraints explicit counter examples for the non-covered cases are rare although Blekherman [2] proved that there must be many of them. On the other side [10] claims that for purposes of optimization, the replacement of the fact that a polynomial is non-negative by the fact that it is a SOS gives very good results in practice. This claim could imply that we can develop an upper bound algorithm for MAXSAT using this SOS approach which gives tighter bounds than the existing ones.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 294–308, 2005.

Let us go into more detail now. Suppose a given polynomial $p(x_1, \dots, x_n) \in \mathbb{R}[x_1, \dots, x_n]$ has to be minimized over $\mathbb{R}^n$. This minimization can be written as the program:

$$\begin{aligned} &\max \alpha \\ &\text{s.t. } p(x_1, \dots, x_n) - \alpha \geq 0 \ \text{ on } \ \mathbb{R}^n \\ &\alpha \in \mathbb{R} \end{aligned} \tag{1}$$

Clearly, the program:

$$\begin{aligned} &\max \alpha \\ &\text{s.t. } \ p(x_1, \dots, x_n) - \alpha \ \text{ is a SOS} \\ &\alpha \in \mathbb{R} \end{aligned} \tag{2}$$

would result in a lower bound for problem (1).

There is a benefit in the approach above using the theory of ‘Newton Polytopes’. The exponent of a monomial $x_1^{a_1} \dots x_n^{a_n}$ is identified with a lattice point $\bar{a} = (a_1, \dots, a_n)$. The Newton polytope associated with a polynomial is the convex hull of all those lattice points associated with monomials appearing in the polynomial involved. Monomials useful for finding a SOS decomposition are those with an exponent $\bar{a}$ for which $2\bar{a}$ is in the Newton polytope. Thus adding more monomials would not enlarge the chance of success. This means that for purposes of global optimization of a polynomial over $\mathbb{R}^n$ we have the advantage to know which monomials are possibly involved in a SOS decomposition (if existing) while we face the disadvantage that non-negative polynomials need not be decomposable as SOS.

Involving the Boolean constraints of the form $x_1^2 - 1 = 0, \dots, x_n^2 - 1 = 0$ the situation turns. It can be proven that polynomials which are non-negative on $\{-1, 1\}^n$ (please note that we use $\{-1, 1\}$- values for Boolean variables instead of the more commonly used $\{0, 1\}$-values, which is much more attractive when algebraic formalisms are involved) can always be written as a SOS modulo the ideal $I_{\mathcal{B}}$ generated by the polynomials $x_1^2 - 1, \dots, x_n^2 - 1$. However, in this case the 'Newton Polytope Property' is not valid, because higher degree monomials may cancel ones with lower degree, when performing calculations modulo $I_{\mathcal{B}}$. Hence, we have to consider possibly an exponential set of monomials in the SOS decomposition. To see this consider a polynomial $p(x_1, \dots, x_n)$ which is non-negative on $\{-1, 1\}^n$. The expression

$$SP(x) = \sum_{\sigma \in \{-1,1\}^n} \frac{p(\sigma)}{2^n} (1 + \sigma_1 x_1) \dots (1 + \sigma_n x_n) \tag{3}$$

is easily seen to give the same outputs on $\{-1, 1\}^n$ as $p(x_1, \dots, x_n)$. Each $\frac{1+\sigma_j x_j}{2}$ is a square modulo $I_{\mathcal{B}}$ because

$$\left(\frac{1 + \sigma_j x_j}{2} \right)^2 \equiv \frac{1 + \sigma_j x_j}{2} \ \text{ modulo } \ I_{\mathcal{B}} \tag{4}$$

Hence, $SP(x)$ is seen to be a SOS modulo $I_{\mathcal{B}}$. At the same time it becomes evident that we might need an exponentially large basis of monomials in realizing this decomposition. We see that if we want to optimize a polynomial over $\{-1,1\}^n$ we have the advantage to know that a basis of monomials exists which will give an exact answer, while we are facing the disadvantage that this basis could be unacceptably large.

We now come to the point of explaining the SOS approach. Let $M^T = (M_1, \ldots, M_k)$ be a row vector of monomials in variables $x_1, \ldots, x_n$ and $p(x_1, \ldots, x_n)$ a given polynomial in $\mathrm{I\!R}[x_1, \ldots, x_n]$. The equation $M^T L^T L M = p$ involving any matrix L of appropriate size would give an explicit decomposition of p as a SOS over the monomials used. Conversely, any SOS decomposition of p can be written in this way. This means that the Semidefinite Program

$$\begin{aligned} &M^T S M = p \\ &S \text{ positive semidefinite } \ (S \succeq 0) \end{aligned} \tag{5}$$

gives a polynomial time decision method for the question whether p can be written as a SOS using $M_1, \ldots, M_k$ as a basis of monomials (up to prescribed precision: the method uses real numbers represented with a certain precision). The constraint $M^T S M = p$ in fact results in a set of linear constraints in the entries of the matrix S.

If we consider the Boolean side constraints we have a similar program. In this case however the equation $M^T S M = p$ needs to be satisfied only modulo $I_{\mathcal{B}}$. Also this constraint results in a set of linear constraints in the entries of the matrix S, but different from the ones above. This is caused by the above mentioned cancellation effects. First we shall associate polynomials to CNF formulae. With a literal X_i we associate the polynomial $\frac{1}{2}(1 - x_i)$ and with $\neg X_j$ we associate $\frac{1}{2}(1 + x_j)$. With a clause we associate the products of the polynomials associated with its literals. Note that for a given assignment $\sigma \in \{-1,1\}^n$ the polynomial associated with each clause outputs a zero or a one, depending of the fact whether σ satisfies the clause or not. With a CNF formula ϕ we associate two polynomials F_ϕ and $F_\phi^{\mathcal{B}}$. F_ϕ is the sum of squares of the polynomials associated with the clauses from ϕ . $F_\phi^{\mathcal{B}}$ is just the sum of those polynomials. Clearly, F_ϕ is non-negative on $\mathrm{I\!R}^n$ and $F_\phi^{\mathcal{B}}$ is non-negative on $\{-1,1\}^n$. $F_\phi(\sigma)$ and $F_\phi^{\mathcal{B}}(\sigma)$ give the number of clauses violated by assignment σ. The following two examples illustrate the construction of F_ϕ and $F_\phi^{\mathcal{B}}$ and the outcome of the corresponding SDP's.

Example 1. Let ϕ be the CNF formula with the following three clauses

$$X \vee Y, \quad X \vee \neg Y, \quad \neg X \tag{6}$$

$$\begin{aligned} F_\phi &= \left(\frac{1}{2}(1-x)\frac{1}{2}(1-y)\right)^2 + \left(\frac{1}{2}(1-x)\frac{1}{2}(1+y)\right)^2 + \left(\frac{1}{2}(1+x)\right)^2 \\ &= \frac{3}{8} + \frac{1}{4}x + \frac{3}{8}x^2 + \frac{1}{8}y^2 + \frac{1}{4}xy^2 + \frac{1}{8}x^2y^2 \end{aligned}$$

In order to attempt to rewrite $F_\phi - \alpha$ as a SOS it suffices to work with the monomial basis $M^T = (1, x, y, xy)$. The program

$$\begin{aligned} &\max \alpha \\ &\text{s.t. } F_\phi - \alpha = M^T S M \\ &\alpha \in \mathrm{I\!R}, S \succeq 0 \end{aligned} \tag{7}$$

gives an output $\alpha = \frac{1}{3}$, from which we may conclude that $2\frac{2}{3}$ is an upper bound for the MAXSAT-solution of ϕ. Notice that $F_\phi = \frac{1}{3} + \frac{3}{8}\left(x + \frac{1}{3}\right)^2 + \frac{1}{8}\left(xy - y\right)^2$. For this ϕ, $F_\phi^{\mathcal{B}} = \frac{1}{2}(1-x)\frac{1}{2}(1-y) + \frac{1}{2}(1-x)\frac{1}{2}(1+y) + \frac{1}{2}(1+x) = 1$. Clearly, $F_\phi^{\mathcal{B}} = 1$ means that any assignment will exactly violate one clause.

Example 2. Let ϕ be the following CNF formula

$$X \vee Y \vee Z, \quad X \vee Y \vee \neg Z, \quad \neg Y \vee \neg T, \quad \neg X, \quad T \tag{8}$$

$$F_\phi^{\mathcal{B}} = \frac{3}{2} + \frac{1}{4}x - \frac{1}{4}t + \frac{1}{4}xy + \frac{1}{4}yt \tag{9}$$

The Semidefinite Program (SDP)

$$\begin{aligned} &\max \alpha \\ &\text{s.t.} F_\phi^{\mathcal{B}} - \alpha \equiv (1, x, y, t) S (1, x, y, t)^T \text{ modulo } I_{\mathcal{B}} \\ &\alpha \in \mathrm{I\!R}, S \succeq 0 \end{aligned} \tag{10}$$

gives output $\alpha = 0.793$, from which we may conclude that 4.207 is an upper bound for the MAXSAT-solution of ϕ. The SDP

$$\begin{aligned} &\max \alpha \\ &\text{s.t.} F_\phi^{\mathcal{B}} - \alpha \equiv (1, x, y, t, xy, xt, yt) S (1, x, y, t, xy, xt, yt)^T \text{ modulo } I_{\mathcal{B}} \\ &\alpha \in \mathrm{I\!R}, S \succeq 0 \end{aligned} \tag{11}$$

gives output $\alpha = 1$. Note that the second SDP gives a tighter upper bound, because more monomials are contained in the basis.

Next, we formulate some useful properties on the polynomials F_ϕ and $F_\phi^{\mathcal{B}}$. Let m be the number of clauses and n the number of variables in ϕ.

Theorem 1. *1. For any assignment $\sigma \in \{-1, 1\}^n, F_\phi(\sigma) = F_\phi^{\mathcal{B}}(\sigma)$. Both give the number of clauses violated by σ.*

2. $\min_{\sigma \in \{-1,1\}^n} F_\phi(\sigma)$ and $\min_{\sigma \in \{-1,1\}^n} F_\phi^{\mathcal{B}}(\sigma)$ give rise to an exact MAXSAT-solution of ϕ: respectively $m - \min_{\sigma \in \{-1,1\}^n} F_\phi(\sigma)$ and $m - \min_{\sigma \in \{-1,1\}^n} F_\phi^{\mathcal{B}}(\sigma)$.

3. $F_\phi^{\mathcal{B}} \equiv F_\phi$ modulo $I_{\mathcal{B}}$.

4. F_ϕ attains its minimum over $\mathrm{I\!R}^n$ somewhere in the hypercube $[-1, 1]^n$ (a compact set), while it can be zero only in a partial satisfying assignment.

5. ϕ is unsatisfiable if and only if there exists an $\epsilon > 0$ such that $F_\phi - \epsilon \geq 0$ on $\mathrm{I\!R}^n$.

6. *If there exists an $\epsilon > 0$ such that $F_\phi - \epsilon$ is a SOS, then ϕ is unsatisfiable.*
7. *If there exists a monomial basis M and an $\epsilon > 0$ such that $F_\phi^{\mathcal{B}} - \epsilon$ is a SOS based on M, modulo $I_{\mathcal{B}}$, then ϕ is unsatisfiable.*
8. *Let M be a monomial basis, then*

$$\begin{aligned} & m - \max \alpha && (12)\\ & s.t.\ \ F_\phi - \alpha \quad is\ a\ SOS \\ & \alpha \in \mathbb{R} \\ & and \\ & m - \max \alpha && (13)\\ & s.t.\ \ F_\phi^{\mathcal{B}} - \alpha \equiv M^T S M \ \ modulo\ \ I_{\mathcal{B}} \\ & \alpha \in \mathbb{R}, S \succeq 0 \end{aligned}$$

give upper bounds for the MAXSAT-solution of ϕ.

Proof. Except for part 1.4 the reasonings behind the other parts are already discussed before or they are direct consequences of earlier statements. Here we prove Theorem 1.4: suppose F_ϕ takes its minimum in x and assume $x_1 = 1+\delta$ for some $\delta > 0$. This gives rise to contributions $(\frac{1}{2}(1+(1+\delta)))^2$ and $(\frac{1}{2}(1-(1+\delta)))^2$. Both contributions are smaller with $\delta = 0$ than with $\delta > 0$. The same argument can be applied for $x_1 = -1-\delta$. Thus $x \in [-1,1]^n$. Furthermore, $F_\phi = 0$ only if in each polynomial associated to a clause at least one of the factors equals zero because F_ϕ is a sum of squares and hence non-negative. This can only be realized if in each polynomial associated to a clause at least one of the variables takes value 1 or -1 resulting in a partial satisfying assignment for ϕ.

Program (13) is the basis for the search for MAXSAT-upper bounds in this paper.

2 Upper Bounds for MAX-2-SAT

Although the SOS approach provides upper bounds for general MAXSAT-solutions, we restrict ourselves in this section to MAX-2-SAT. The reason is that we want to present a comparison with the results of the famous methods of Goemans and Williamson [8] and Feige and Goemans [6].

The first monomial basis we consider in the SOS approach (13) is $M^{GW} = (1, x_1, \ldots, x_n)$, where the x_i's come from the variables in ϕ.

Semidefinite Programming formulations come in pairs: the so-called primal and dual formulations, see for example [5] and [4]. Here we present a generic formulation of a primal semidefinite problem P, and its dual program D. In our context, D and P give the same optimal value.

Consider the program P

$$\min \operatorname{Tr}(CX) \tag{14}$$

$$\text{s.t. } \operatorname{diag}(X) = e \tag{15}$$

$$\begin{aligned} &\mathrm{Tr}(A_j X) \geq 1, \quad j = 1, \dots, k \qquad (16)\\ &X \succeq 0 \end{aligned}$$

In the above formulation, C and X are symmetric square matrices of size, say $p \times p$. Tr means the trace of the matrix, i.e. the sum of the entries on the diagonal.

$$Tr(CX) = \sum_{i=1}^{p} \sum_{j=1}^{p} c_{ij} x_{ij}$$

diag$(X) = e$ means that the entries on the diagonal of matrix X are all ones. The A_j's are square symmetric matrices.

The program P has the following dual program D

$$\begin{aligned} &\max \sum_{i=1}^{p} \gamma_i + \sum_{j=1}^{k} y_j \qquad (17)\\ &\text{s.t.Diag}(\gamma) + \sum_{j=1}^{k} y_j A_j + U = C\\ &U \succeq 0, y_j \geq 0 \end{aligned}$$

in which the y_j's are the dual variables corresponding to constraints (16) and the γ_i's the dual variables corresponding to constraints (15). U is a symmetric square matrix and Diag(γ) is the square matrix with the γ_i's on the diagonal and all off-diagonal entries equal to zero.

2.1 Comparison of SOS Approach and Goemans-Williamson Approach

The original Goemans-Williamson approach for obtaining an upper bound for a MAX-2-SAT problem starts with $F_\phi^{\mathcal{B}}$ too. The problem

$$\begin{aligned} &\min F_\phi^{\mathcal{B}}(x) \qquad (18)\\ &x \in \{-1, 1\}^n \end{aligned}$$

is relaxed by relaxing the Boolean arguments x_i. With each x_i, a vector $v_i \in \mathrm{I\!R}^{n+1}$ is associated, with norm 1, and products $x_i x_j$ are interpreted as inproducts $v_i \bullet v_j$. In this way, they turn (18) into a Semidefinite Program, after making $F_\phi^{\mathcal{B}}$ homogenous by adding a dummy vector v_{n+1} in order to make the linear terms in $F_\phi^{\mathcal{B}}$ quadratic as well. For example, $3x_i$ is replaced by $3(v_i \bullet v_{n+1})$. Let $\hat{F}_\phi^{\mathcal{B}}$ be the polynomial constructed from $F_\phi^{\mathcal{B}}$ in this way. The problem Goemans and Williamson solve is

$$\begin{aligned} &\min \hat{F}_\phi^{\mathcal{B}}(v_1, \dots, v_n, v_{n+1}) \qquad (19)\\ &s.t. \ \|v_i\| = 1, v_i \in \mathrm{I\!R}^{n+1} \end{aligned}$$

To transform (19) to a semidefinite program, $v_i \bullet v_j$ is replaced by t_{ij}. Let b_{ij} be the coefficient of $M_i^{GW} M_j^{GW}$ in the polynomial $F_\phi^{\mathcal{B}}$. Let $f_{ij} = f_{ji} = \frac{1}{2} b_{ij}$ and $f_{ii} = 0$ for each i. Let $M(F)$ be the symmetric matrix with entries f_{ij} and T is a symmetric matrix of the same size. Furthermore, let c_0 be the constant term in $F_\phi^{\mathcal{B}}$. To be precise, c_0 equals $\frac{1}{2}$ times the number of 1-literal clauses plus $\frac{1}{4}$ times the number of 2-literal clauses in ϕ.

Consequently, (19) is equivalent to the following semidefinite program

$$\begin{aligned} &c_0 + \min \sum_{i=1}^{n+1} \sum_{j=1}^{n+1} f_{ij} t_{ij} \\ &\text{s.t. } \; t_{ii} = 1, T \succeq 0 \end{aligned} \tag{20}$$

or in matrix notation,

$$\begin{aligned} &c_0 + \min \operatorname{Tr}(M(F)T) \\ &\text{s.t. } \; \operatorname{diag}(T) = e, T \succeq 0 \end{aligned} \tag{21}$$

While Goemans and Williamson relax the input arguments of $F_\phi^{\mathcal{B}}$, the SOS-method is a relaxation by replacing non-negativity by being a SOS. The next theorem proves that the SOS-approach with monomial basis M^{GW} gives the same upper bound for MAX-2-SAT as program (21).

Theorem 2. *The SOS-approach with monomial basis M^{GW} gives the same upper bound as the upper bound obtained by the algorithm by Goemans and Williamson.*

Proof. In the Goemans-Williamson SDP (21) we only have the constraints of type (15), and not of type (16). This implies that we have to deal only with the γ_j-variables. The size of the variable matrix T is $n+1$, with n the number of variables in the CNF formula.

The dual problem of the GW-semidefinite program (21) is

$$\begin{aligned} &c_0 + \max \sum_{i=1}^{n+1} \gamma_i \\ &\text{s.t. } \; \operatorname{Diag}(\gamma) + U = M(F) \\ &U \succeq 0, \gamma_i \; \text{ free} \end{aligned} \tag{22}$$

We start with the program

$$\begin{aligned} &\max \alpha \\ &\text{s.t. } F_\phi^{\mathcal{B}} - \alpha \equiv M^T S M \; \text{ modulo } \; I_{\mathcal{B}} \\ &S \succeq 0, \alpha \in \mathbb{R} \end{aligned} \tag{23}$$

with monomial basis $M = M^{GW} = (M_1^{GW}, \dots, M_{n+1}^{GW}) = (1, x_1, \dots, x_n)$ and prove that it is equal to (22). Let s_{ij} be the (i,j)-th element of matrix S.

We can reformulate program (23) as

$$\begin{aligned}
&\max \alpha && (24)\\
&\text{s.t.} \sum_{i=1}^{n+1}\sum_{j=1}^{n+1} s_{ij} M_i^{GW} M_j^{GW} \equiv F_\phi^{\mathcal{B}} - \alpha \ \text{ modulo } \ I_{\mathcal{B}} &&\\
&S \succeq 0, \alpha \in \mathbb{R} &&
\end{aligned}$$

Consider the constraint in program (24) for the coefficient of the constant. On the left hand side we have $\sum_{i=1}^{n+1} s_{ii}$, on the right hand side we have $c_0 - \alpha$. This results in the equality

$$\alpha = c_0 - \sum_{i=1}^{n+1} s_{ii} \tag{25}$$

We transform (24) to matrix notation. In the matrix formulation (26), on both left and right hand sides we have a matrix with on position (i, j), $i \neq j$, the coefficient of $M_i^{GW} M_j^{GW}$ using symmetry. Substituting (25) and using matrix notation we can reformulate (24) as

$$\begin{aligned}
&c_0 + \max \sum_{i=1}^{n+1} -s_{ii} && (26)\\
&\text{s.t. } \ S - \operatorname{diag}(S) = M(F) &&\\
&S \succeq 0 &&
\end{aligned}$$

Identifying γ_i with matrix entries $-s_{ii}$ and U with S, it is immediate that (26) is equivalent to (22).

Hence, we proved that (23) with monomial basis M^{GW} equals (22). It can be concluded that the Goemans-Williamson SDP and the SDP of the SOS-approach with monomial basis M^{GW} are dual problems providing the same upper bound to the MAX-2-SAT solution.

Still, there is something more to say about these two different approaches. (20) has $\frac{1}{2}(n+1)(n+2)$ variables t_{ij} (not $(n+1)^2$ because T is symmetric). In the SOS approach (13) with monomial basis M^{GW}, it is not hard to see that in fact only the diagonal elements are essentially variable, because the off-diagonal elements are fixed. This means that the actual dimension of the SOS-program with monomial basis M^{GW} is linear in the number of variables, while in the Goemans-Williamson formulation (20) the dimension grows quadratically.

In our experiments we tried several semidefinite programming solvers like Sedumi [11], DSDP [1], CSDP [3] and SDPA [7], but none of them could fully benefit from this factHowever, we found that CSDP performed best on SDP's of the form (13).

2.2 Adding Valid Inequalities vs Adding Monomials

Feige and Goemans [6] propose to add so-called valid inequalities to (21) in order to improve the quality of the relaxation. A valid inequality is an inequality that is satisfied by any optimal solution os the original (unrelaxed) problem but may be violated by the optimal solution of the relaxation. Valid inequalities improve the quality of the relaxation because they exclude a part of its feasible region that cannot contain the optimal solution of the original problem. Triangle inequalities are among the most frequently used valid inequalities. Two types of these 'triangle inequalities' are considered. The first is the inequality

$$1 + x_i + x_j + x_i x_j \geq 0 \tag{27}$$

Note that in (21) t_{ij} has replaced $x_i x_j$ and $t_{i,n+1}$ replaces x_i from $F_\phi^\mathcal{B}$. In fact, they consider the homogeneous form $1 + t_{i,n+1} + t_{j,n+1} + t_{i,j} \geq 0$ which is added to (21). One might add all these inequalities, also the ones obtained by replacing x_i and/or x_j by $-x_i$ or $-x_j$. Another possibility is to add only those inequalities where X_i and X_j appear together in the same clause. In this section, we examine how this compares with the SOS approach. It can be shown that $1+x_i+x_j+x_i x_j$ cannot be recognized as a SOS based on $M = (1, x_i, x_j)$. However, if we add $x_i x_j$ to the monomial basis we have

$$1 + x_i + x_j + x_i x_j \equiv \frac{1}{4}\left(1 + x_i + x_j + x_i x_j\right)^2 \;\text{ modulo }\; I_\mathcal{B}$$

A similar argument can be given for the three inequalities with x_i and/or x_j replaced by $-x_i$ and/or $-x_j$. Hence, the effect of adding the valid inequality (27) and the three similar inequalities is captured in the SOS approach by adding the monomial $x_i x_j$ to the basis. Below we prove a theorem from which follows that adding the monomial $x_i x_j$ results in upper bounds that are at least as tight as the upper bound of the Feige-Goemans program with the four triangle inequalities of the form (27). The experiments in section 2.4 support this fact.

Feige and Goemans [6] further showed that adding for each triple of variables X_i, X_j and X_k to (21) the valid inequalities

$$\begin{aligned} 1 + t_{ij} + t_{ik} + t_{jk} \geq 0, &\quad 1 - t_{ij} + t_{ik} - t_{jk} \geq 0 \\ 1 + t_{ij} - t_{ik} - t_{jk} \geq 0, &\quad 1 - t_{ij} - t_{ik} + t_{jk} \geq 0 \end{aligned} \tag{28}$$

improves the tightness of the relaxation. Note that

$$1 + x_i x_j + x_i x_k + x_j x_k \equiv \frac{1}{4}\left(1 + x_i x_j + x_i x_k + x_j x_k\right)^2 \;\text{ modulo }\; I_\mathcal{B}$$

Hence, the effect of adding the four inequalities (28) is captured by adding $x_i x_j$, $x_i x_k$ and $x_j x_k$ to the monomial basis. Also in this case, the effect of adding the monomials to the monomial basis results in upper bounds at least as tight compared to adding valid inequalities to the Goemans-Williamson SDP as shown in Theorem 3.

Finally, note that adding all inequalities of the form (28) amounts to adding $\mathcal{O}(n^3)$ inequalities, while in the SOS approach $\mathcal{O}(n^2)$ monomials of degree 2 need to be added. For the moment it is too early to decide whether existing SDP-solvers are suitable, or can be modified, to turn this effect into a computational benefit as well.

Theorem 3. *Adding monomials x_ix_j, x_ix_k and x_jx_k to the monomial basis in the SOS approach gives an upper bound at least as tight as the upper bound obtained by adding triangle inequalities of the type (28) to the Goemans-Williamson SDP.*

Proof. Without loss of generality we consider the triangle inequality

$$1 + x_1x_2 - x_1x_3 - x_2x_3 \geq 0 \tag{29}$$

In the notation of (20) this equation is $1+t_{12}-t_{13}-t_{23} \geq 0$. In matrix notation this inequality is $Tr(AT) \geq 1$ with

$$A = \begin{pmatrix} 1 & 1 & -1 \\ 1 & 1 & -1 \\ -1 & -1 & 1 \end{pmatrix}$$

We consider the program

$$\begin{aligned} &\min \mathrm{Tr}(M(F)T) \\ &\text{s.t. } \mathrm{diag}(T) = e \\ &\mathrm{Tr}(AT) \geq 1 \\ &T \succeq 0 \end{aligned} \tag{30}$$

Assume that F is an homogenous polynomial of degree 2 in three variables x_1, x_2, x_3 only. This does not harm the general validity of this proof but makes the key steps more transparent. $M(F)$ is the coefficient matrix associated with the polynomial F. Let $F(x_1, x_2, x_3) = 2ax_1x_2 + 2bx_1x_3 + 2cx_2x_3$. Then,

$$M(F) = \begin{pmatrix} 0 & a & b \\ a & 0 & c \\ b & c & 0 \end{pmatrix}$$

The dual program of (30) is the following

$$\begin{aligned} &\max \gamma_1 + \gamma_2 + \gamma_3 + y \\ &\text{s.t. } yA + \mathrm{Diag}(\gamma) + U = M(F) \\ &U \succeq 0, y \geq 0 \end{aligned} \tag{31}$$

with Diag(γ) the 3×3-matrix with on its diagonal $\gamma_1, \gamma_2, \gamma_3$.

Program (31) can be reformulated as

$$\begin{aligned} &\max \gamma_1 + \gamma_2 + \gamma_3 + y \\ &\text{s.t. } M(F) - \mathrm{Diag}(\gamma) - yA \succeq 0 \\ &y \geq 0 \end{aligned} \tag{32}$$

Now suppose that $(\hat{\gamma_1}, \hat{\gamma_2}, \hat{\gamma_3}, \hat{y})$ is an optimal solution for (32). We will show that from this optimal solution a feasible solution for (33) can be constructed with $M = (1, x_1, x_2, x_3, x_1x_2, x_1x_3, x_2x_3)$. In fact, we will even show that the monomial basis $M_1 = (1, x_1x_2, x_1x_3, x_2x_3)$ is already sufficient in this respect.

$$\begin{aligned} &\max \alpha \\ &\text{s.t. } MSM^T \equiv F - \alpha \quad \text{modulo } I_{\mathcal{B}} \end{aligned} \tag{33}$$

Program (33) with monomial basis M_1 can be reformulated as

$$\begin{aligned} &\max \left(-\sum_{i=1}^{4} s_{ii} \right) \\ &s_{12} + s_{21} + s_{34} + s_{43} = 2a \\ &s_{13} + s_{31} + s_{24} + s_{42} = 2b \\ &s_{23} + s_{32} + s_{14} + s_{41} = 2c \\ &S \succeq 0 \end{aligned} \tag{34}$$

$1 + x_1x_2 - x_1x_3 - x_2x_3$ is a SOS modulo $I_{\mathcal{B}}$, because the following holds

$$1 + x_1x_2 - x_1x_3 - x_2x_3 = \frac{1}{4} M_1 \Delta M_1^T \tag{35}$$

with Δ the positive semidefinite matrix

$$\Delta = \begin{pmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \end{pmatrix}$$

Let Z be the 4×4 matrix with the 3×3 matrix $M(F) - \text{Diag}(\hat{\gamma}) - \hat{y}A$ starting in the upper left corner and having zeros in fourth row and column. We can conclude that $Z + \frac{1}{2}\hat{y}\Delta \succeq 0$ because $Z \succeq 0$, $\Delta \succeq 0$ and $\hat{y} \geq 0$. The matrix $Z + \frac{1}{2}\hat{y}\Delta$ satisfies the constraints in (34) and $-\sum_{i=1}^{4} s_{ii} = \hat{\gamma_1} + \hat{\gamma_2} + \hat{\gamma_3} + \hat{y}$.

Because the optimal solution of Feige-Goemans SDP with valid inequalities equals a feasible solution of the SDP approach with monomials of degree 2 in the monomial basis, it can be concluded that the SOS approach gives at least as tight upper bounds.

2.3 SOS-Approach on Feige-Goemans Example

In general the CNF formula of $dFGn$ in n variables is defined as

$$\begin{aligned} &x_1 \vee x_2,\ x_2 \vee x_3,\ x_3 \vee x_4,\ \ldots, x_n \vee x_1 \\ &\neg x_1 \vee \neg x_2,\ \neg x_2 \vee \neg x_3,\ \neg x_3 \vee \neg x_4,\ \ldots,\ \neg x_n \vee \neg x_1 \end{aligned} \tag{36}$$

Feige and Goemans [6] present $dFG5$ as worst-known case example with respect to the performance guarantee of their approach.

Note that we can satisfy $2n-1$ of the clauses if n is odd by setting the odd-numbered variables to true and the even-number variables to false. It is not possible to satisfy all clauses for odd n. In this section, we show that the SOS-approach with a monomial basis 1, all variables and all possible monomials of degree 2 solves (36) to optimality.

Theorem 4. *Let n be an odd number. The polynomial $F^{\mathcal{B}}_{\phi_n} - 1$ with*

$$F^{\mathcal{B}}_{\phi_n} = \frac{1}{2}\left(n + x_1x_2 + x_2x_3 + \ldots + x_{n-1}x_n + x_nx_1\right) \tag{37}$$

is a sum of squares if we choose as monomial basis $M = \{1, x_1, \ldots, x_n\}$ extended with the set of all possible monomials of degree 2.

Proof. As initial step we start with $dFG3$. The polynomial $F^{\mathcal{B}}_{\phi_3}$ is

$$F^{\mathcal{B}}_{\phi_3} = \frac{1}{2}\left(3 + x_1x_2 + x_2x_3 + x_3x_1\right) \tag{38}$$

Notice that $F^{\mathcal{B}}_{\phi_3} - 1$ is a sum of squares modulo $I_{\mathcal{B}}$, because

$$\frac{1}{2}\left(\frac{1}{2}\left(1 + x_1x_2 + x_2x_3 + x_3x_1\right)\right)^2 \equiv \frac{1}{2}\left(1 + x_1x_2 + x_2x_3 + x_3x_1\right)$$

We use this fact to prove by induction that $F^{\mathcal{B}}_{\phi_n} - 1$ is also a sum of squares relative to the monomial basis considered. Assume that the polynomial $F^{\mathcal{B}}_{\phi_{n-2}} - 1$ related to $dFG(n-2)$ is a sum of squares modulo $I_{\mathcal{B}}$.

The polynomial $F^{\mathcal{B}}_{\phi_n}$ equals

$$F^{\mathcal{B}}_{\phi_n} = F^{\mathcal{B}}_{\phi_{n-2}} + \frac{1}{2}(2 + x_{n-2}x_{n-1} + x_{n-1}x_n + x_nx_1 - x_{n-2}x_1) \tag{39}$$

We assumed that $F^{\mathcal{B}}_{\phi_{n-2}} - 1$ is a sum of squares. Let $T_1(x) = 1 - x_1x_{n-2} + x_{n-2}x_{n-1} + x_1x_{n-1}$ and $T_2(x) = 1 - x_1x_{n-1} + x_{n-1}x_n + x_nx_1$. Note that $F^{\mathcal{B}}_{\phi_n} = F^{\mathcal{B}}_{\phi_{n-2}} + \frac{1}{2}(T_1(x) + T_2(x))$ and for $i = 1$ and $i = 2$

$$\frac{1}{2}\left(\frac{1}{2}T_i(x)\right)^2 \equiv \frac{1}{2}T_i(x) \ \text{ modulo } \ I_{\mathcal{B}}$$

This proves that $F^{\mathcal{B}}_{\phi_n} - 1$ is also a sum of squares.

From this theorem we can conclude that the SOS-approach with monomial basis 1, the variables and all possible monomials of degree 2 identifies $F^{\mathcal{B}}_{\phi_n} - 1$ as a sum of squares. Hence, the minimum of $F^{\mathcal{B}}_{\phi_n}$ is at least 1. We conclude that our SOS approach solves (36) to optimality.

2.4 Experimental Results

In this section we consider besides the Goemans-Williamson upper bound the next four variants of the Feige-Goemans method.

Variant FG_m: The valid inequalities added in this variant are only those coming directly from the clauses. For instance, if $X \vee \neg Y$ is a clause, we add the valid inequality $1 - x + y - xy \geq 0$.
Variant FG_{4p}: For each pair of variables X_i and X_j occurring in a same clause, the four inequalities of the type (27) are added.
Variant FG_{ap}: For *each* pair of variables the four inequalities of the type (27) are added.
Variant FG_{pt}: All inequalities of variant FG_{ap} are added and additionally for each triple of variables the four inequalities of type (28) added.

We compare the upper bounds resulting from these variants with the upper bound obtained from the semidefinite program (13) with monomial basis M_p consisting of the set $\{1, x_1, \ldots, x_n\}$ extended with all $x_i x_j$ for variables X_i and X_j appearing in a same clause. We call the corresponding upper bound SOS_p. SOS_{ap} is the variant with monomial basis $\{1, x_1, \ldots, x_n\}$ extended with all $x_i x_j$ for each pair of variables X_i and X_j. We will restrict ourselves to small-scale problems in this section, because solving the SDP's of SOS_p takes a lot of time with the SDP-solvers currently available.

In our initial experiments we used a set of 900 randomly generated instances with 10 variables and different densities. For each of the densities 1.0, 1.5, 2.0, ..., 5.0, 100 instances are considered. The 'bound ratio' R is defined as the optimal MAXSAT solution divided by the upper bound found. In Table 1 we give for each method the average R over the set of unsatisfiable instances out of the 100 generated instances for each density. In Table 1, the first column indicates the density, the second the upper bounds by SOS_p, the third column gives the results of SOS_{ap}, the fourth gives the Goemans-Williamson upper bound, the fifth gives the upper bound of variant FG_m, the next the upper bound by variant FG_{4p}, then the upper bound of FG_{ap} and the last column gives upper bound of variant FG_{pt}.

From Table 1 we see that the upper bounds obtained by SOS_{ap} are at least as tight as the other ones. This is not only true on average but in fact for each

Table 1. 10 variables, MAX-2-SAT

d	SOS_p	SOS_{ap}	GW	FG_m	FG_{4p}	FG_{ap}	FG_{pt}
1.0	1	1	0.933480	0.984195	0.993151	0.993151	1
1.5	1	1	0.953544	0.989779	0.993869	0.993869	1
2.0	0.99984	1	0.969460	0.994136	0.997043	0.997242	1
2.5	1	1	0.979549	0.996760	0.998317	0.998420	1
3.0	1	1	0.981904	0.996485	0.998044	0.998188	1
3.5	1	1	0.985519	0.997435	0.998904	0.998975	1
4.0	0.999995	1	0.987250	0.997538	0.998873	0.998930	0.999964
4.5	0.999973	0.999973	0.986327	0.997120	0.998692	0.998776	0.999936
5.0	0.999979	0.999979	0.987015	0.997779	0.998764	0.998841	0.999971

Table 2. 25 variables, MAX-2-SAT

d	SOS_p	GW	FG_{ap}
1.5	0.999403	0.955241	0.995686
2.0	0.999607	0.968157	0.997279
2.5	0.999577	0.976133	0.997639
3.0	0.999906	0.979769	0.998238
3.5	0.999691	0.981444	0.997520
4.0	0.999908	0.982812	0.997890
4.5	0.999882	0.983297	0.997721
5.0	0.999989	0.984816	0.998196

individual instance. SOS_p is almost always, except for one instance, at least as good as the best Feige-Goemans variant FG_{pt}. In these experiments with MAX-2-SAT instances with 10 variables, the SDP's of each variant are solved by Sedumi [11]. Table 2 gives the same type of results for instances with 25 variables but only for the upper bounds that are most relevant and computationally not too expensive to obtain. For the instances with 25 variables GW and FG_{ap} are solved by Sedumi. The SDP's of SOS_p are solved by CSDP [3], because this solver is faster, more accurate and uses less memory when solving these SDP's.

2.5 Note on the Complexity of Different Approaches

The complexity of short step semidefinite optimization algorithms (like for example Sedumi) is $\mathcal{O}((2V^2+C)\sqrt{V})$ if V is the size of the semidefinite variable-matrix in the SDP and C the number of constraints. We will compare the computational complexity of the different upper bound variants in this section. Let ϕ be a CNF formula with n variables and m clauses.

For each of the variants considered the V's and C's involved lead to the following complexities

$$CP_{GW} = \mathcal{O}(n^2\sqrt{n}) \tag{40}$$

$$CP_{FG_m} = \mathcal{O}((n^2+m)\sqrt{n}) \tag{41}$$

$$CP_{FG_{4p}} = \mathcal{O}((n^2+m)\sqrt{n}) \tag{42}$$

$$CP_{FG_{ap}} = \mathcal{O}\left(n^2\sqrt{n}\right) \tag{43}$$

$$CP_{FG_{pt}} = \mathcal{O}(n^3\sqrt{n}) \tag{44}$$

$$CP_{SOSp} = \mathcal{O}\left((n+m)^2\sqrt{n+m}\right) \tag{45}$$

$$CP_{SOSap} = \mathcal{O}(n^5) \tag{46}$$

3 Conclusions and Future Research

In this paper, we presented a new approach for computing upper bounds for MAX-SAT. We show theoretically and experimentally that it gives for MAX-2-SAT at least as tight upper bounds as the approaches by Goemans and Williamson [8] and Feige and Goemans [6].

In a next paper, we conclude that a combination of the original rounding procedures of Goemans and Williamson and of Feige and Goemans, which provide the lower bounds they use to obtain the performance ratio guarantee of their methods, with the SOS based upper bound techniques we proposed, leads to polynomial time algorithms for MAX-2-SAT having performance ratio guarantee at least as good, but observably better in particular cases. A similar conclusion can be drawn with respect to the Karloff and Zwick results for MAX 3-SAT.

Future research should mainly concentrate on developing SDP software which is specifically designed for dealing with the type of problems emerging from the SOS approach. They possess a very special structure which could be explored.

References

[1] S.J. Benson and Y. Ye. DSDP5 User guide - The dual-scaling algorithm for semidefinite programming. Technical Report ANL/MCS-TM-255, Argonne National Laboratory, 2005.

[2] G. Blekherman. There are significantly more nonnegative polynomials than sums of squares. submitted to Israel Journal of Mathematics, 2004.

[3] B. Borchers. CSDP : A C library for semidefinite programming. Technical report, New Mexico Tech, 1997.

[4] E. de Klerk. *Aspects of Semidefinite Programming: Interior Point Algorithms and Selected Applications*, volume 65 of *Applied Optimization Series.* Kluwer Academic Publishers, 2002.

[5] E. de Klerk and J.P. Warners. Semidefinite programming relaxations for MAX 2-SAT and 3-SAT: Computational perspectives. In *Combinatorial and Global Optimization, P.M. Pardalos, A. Migdalas, and R.E. Burkard (eds.), Series on Applied Optimization, Volume 14.* World Scientific Publishers, 2002.

[6] U. Feige and M.X. Goemans. Approximating the value of two prover proof systems, with applications to MAX2SAT and MAXDICUT. In *Proceedings of theThird Israel Symposium on Theory of Computing and Systems*, pages 182–189, 1995.

[7] K. Fujisawa, M. Kojima, K. Nakata, and M. Yamashita. SDPA(Semidefinite Programming Algorithm) : user's manual. Research Reports on Information Sciences, Ser. B : Operations Research B308, Dept. of Information Sciences, Tokyo Institute of Technology, 2-12-1, Oh-Okayama, Meguro-ku, Tokyo 152, Japan, 2002.

[8] M.X. Goemans and D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the Assoc. Comput. Mach.*, 42(6):1115–1145, 1995.

[9] H. Karloff and U. Zwick. A 7/8-approximation algorithm for MAX 3SAT? In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, Miami Beach, FL, USA.* IEEE Press, 1997.

[10] P.A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming Ser. B*, 96(2):293–320, 2003.

[11] J.F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11–12:625–653, 1999.

Faster Exact Solving of SAT Formulae with a Low Number of Occurrences per Variable

Magnus Wahlström[*]

Department of Computer and Information Science,
Linköping University,
SE-581 83 Linköping, Sweden
magwa@ida.liu.se

Abstract. We present an algorithm that decides the satisfiability of a formula F on CNF form in time $O(1.1279^{(d-2)n})$, if F has at most d occurrences per variable or if F has an average of d occurrences per variable and no variable occurs only once. For $d \leq 4$, this is better than previous results. This is the first published algorithm that is explicitly constructed to be efficient for cases with a low number of occurrences per variable. Previous algorithms that are applicable to this case exist, but as these are designed for other (more general, or simply different) cases, their performance guarantees for this case are weaker.

1 Introduction

The boolean satisfiability problem, and its restricted variants, is one of the most well studied classes of NP-complete problems. Since no algorithm for general formulae on conjunctive normal form (CNF) with a worst-case running time of $O(c^n)$ for $c < 2$ is known, or even believed to exist (the currently fastest algorithms for SAT run in $O(2^{n(1-1/\log_2 2m)})$ time [6, 15] where m is the number of clauses and expected time $O(2^{n-c\sqrt{n}})$ for a constant c [5]), a large amount of work has been done on restricted variants of the problem that are easier to solve. Most notable of these restricted problems is k-SAT where a clause may have at most k literals. This is polynomial for $k = 2$ and NP-complete for $k > 2$ [9]. The best results for 3SAT are a probabilistic algorithm which runs in time $O(1.3238^n)$ [11] and a deterministic one which runs in time $O(1.473^n)$ [1], and for general k-SAT a probabilistic algorithm with running time in $O((2 - 2/k)^n)$ [14] and a deterministic algorithm in time $O((2 - 2/(k + 1))^n)$ [4].

If every variable is limited to at most d occurrences in a formula, SAT is solvable in linear time when $d = 2$ and NP-complete when $d \geq 3$, and k-SAT where every clause has exactly k literals is trivial when $d \leq k$ (every such formula is satisfiable) and NP-complete otherwise. If shorter clauses are allowed, 3SAT is NP-complete when $d \geq 3$ [17]. Previous comparable algorithms include an

[*] This research work was funded by CUGS (the national graduate school in computer science).

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 309–323, 2005.

unpublished result by Kullmann (cited in [13]), where he achieves $O(3^{n/9}) \approx O(1.1299^n)$ for SAT instances where $d = 3$, and two algorithms with running times characterised by measures other than n that give non-trivial results for some of these cases. Hirsch has given algorithms [10] that run in time $O(1.2389^K)$ and $O(1.0740^L)$ for a general SAT formula with K clauses or total length L, which would give $O(1.2389^n)$ for $d = 3$ and $O(1.3305^n)$ for $d = 4$, and Szeider has given a fixed-parameter tractable algorithm with a running time characterised by the maximum deficiency of a formula F [16]. If $K(F)$ is the number of clauses in F and $N(F)$ the number of variables, the maximum deficiency of F is $D = \max_{F'}(K(F') - N(F'))$ over every subformula F' of F, and Szeider's algorithm runs in time $O(2^D n^4)$. For a k-SAT instance where every clause has exactly k literals, and with $d \geq k$, this would give us a running time of $O(2^{(d/k-1)n})$, or $O(1.2600^n)$ when $d = 4$ and $k = 3$ (or indeed for any formula F where $d = 4$ and where no clause has fewer than three literals). When 2-clauses are allowed, the result is not as strong. In this paper, we present an algorithm that runs in time $O(1.1279^{(d-2)n})$ for any CNF formula with at most d occurrences per variable, *i.e.* $O(1.1279^n)$ when $d = 3$ and $O(1.2721^n)$ when $d = 4$, regardless of the lengths of the clauses. In a slight variation of this, we get the same running time if a formula F is free of singletons (variables with one occurrence) and the average number of occurrences per variable is d. Hirsch's results hold for this case as well. This latter case is perhaps of less complexity theoretical interest, but might occur in practice, if one has an application that creates formulae with few high-degree variables. The reason that we cannot allow singletons in the latter case is that they lower the average degree too much. While all singletons disappear in reductions without adding any significant extra running time, the value of the function used as an upper bound on the running time, which is a function of $(d - 2) \cdot n$, actually decreases when singletons are added–in the extreme case, one could add singletons until $d < 2$ and the upper bound would collapse. It would be possible to compensate for this by introducing extra terms in the function, but this would make the entire analysis more complex. We feel that the restriction, which only applies when one is counting the average number of occurrences per variable, is not severe enough to warrant this.

The results in this paper are achieved by a compact recursive algorithm with about a dozen cases. The strategy used in the algorithm is to use reductions and branchings to gradually impose more structure onto the problem instance, until finally the problem is structured enough to be converted into an instance of $(3, 2)$-CSP with a third as many variables. This instance can then be solved by a fast algorithm for this problem, such as Eppstein's algorithm [8]. In particular, we found it possible to enforce a large amount of structure when every variable occurs at most three times in F. Another key component is our measure of complexity $f(F) = L(F) - 2N(F)$, which we use to guide the decision of when to apply certain reductions. It is this measure that allows us to construct an algorithm that is simultaneously very strong when every variable occurs at most three times, and able to handle instances where some variables have a higher number of occurrences well.

The structure of this paper is as follows. Section 2 contains some notes on standard concepts used in the text, Section 3 presents the algorithm, Section 4 contains the proof of its running time, and finally Section 5 contains conclusions and some discussion of future work.

2 Preliminaries

A SAT instance is a boolean formula F in CNF form, with no restrictions on the clause lengths. A k-SAT instance is a SAT instance where no clause contains more than k literals; a clause with exactly k literals is a k-clause. The problem instances that are considered in this paper belong to the general SAT class, without restrictions on clause lengths. $Vars(F)$ is the set of all variables that occur in F.

Regarding notation, if v is a variable then $\neg v$ is its negative literal, and if l is the literal $\neg v$ then $\neg l$ is the literal v. In general, l (or l', l_i, etc) represents a literal that may be positive or negative, while other lowercase letters represent variables or positive literals. $|C|$ for a clause C denotes the number of literals in C (also called the length of C), and a clause $(l \vee C)$ for some literal l and clause C is the clause that contains all literals of C plus the literal l. Similarly, $(l \vee C \vee D)$ for literal l and clauses C and D would be the clause containing l plus every literal occurring in C or D. Unless explicitly stated otherwise, the clauses C, D must be non-empty.

If l is a literal of a variable occurring in F, $F[l]$ is the formula one gets if every clause C in F that contains l, and every occurrence of $\neg l$ in other clauses, is removed. For a set of literals A, $F[A]$ is the result of performing the same set of operations for every literal $l' \in A$. Note that $F[l]$ and $F[A]$ may contain empty clauses.

For a variable $v \in Vars(F)$, define the *degree* $d(v, F)$ of v in F to be the number of occurrences of either v or $\neg v$ in the clauses of F. Usually, the formula is understood from the context, and $d(v)$ is used. $d(F)$ is the maximum degree of any $v \in Vars(F)$, and F is *k-regular* if $d(v) = k$ for every $v \in Vars(F)$. A variable v where the literal v occurs in a clauses and $\neg v$ occurs in b clauses is an (a, b)-variable, in which case v is an a-literal and $\neg v$ a b-literal. If $b = 0$, then v is a *pure literal* (similarly, if $a = 0$ then $\neg v$ is a pure literal). We will usually assume, by symmetry, that $a \geq b$, so that *e.g.* any 1-literal is always a negative literal $\neg v$. If $d(v) = 1$, then v is a *singleton*.

The *neighbours* of a literal l are all literals l' such that some clause C in F contains both l and l'. If a clause C contains a literal of both variables a and b, then a and b *co-occur* in C.

We write $L(F)$ for the length of F and $N(F)$ for the number of variables in F (*i.e.* $L(F) = \sum_{v \in Vars(F)} d(v, F)$ and $N(F) = |Vars(F)|$).

We use the classic concept of *resolution* [7]. For clauses $C = (a \vee l_1 \vee \ldots \vee l_d)$ and $D = (\neg a \vee m_1 \vee \ldots \vee m_e)$, the *resolvent* of C and D by a is the clause $(l_1 \vee \ldots \vee l_d \vee m_1 \vee \ldots \vee m_e)$, shortened to remove duplicate literals. If this new clause contains both v and $\neg v$ for some variable v, it is said to be a *trivial resolvent*.

For a formula F and a variable v occurring in F, $DP_v(F)$ is the formula where all non-trivial resolvents by v have been added to F and all clauses containing the variable v have been removed from F. Resolution is the process of creating $DP_v(F)$ from F.

(d,l)-CSP is the constraint satisfaction problem where each variable has d possible values and every constraint involves l variables. $(3,2)$-CSP is used in a subcase of the algorithm in this paper, and for this problem there is an algorithm due to Eppstein [8] which runs in $O(1.3645^n)$ time for n variables. The problem formulation we use is the one used by Eppstein: An instance I of (d,l)-CSP is a collection of variables and a collection of constraints. For each variable v, there is a list of up to d values (called colours) that v can take, and each constraint is a tuple of up to l (variable,colour)-pairs. The constraints are seen as illegal combinations: A constraint $((v_1,X_1),\ldots,(v_k,X_k))$ is satisfied if, for at least one i, $1 \leq i \leq k$, variable v_i is assigned a colour different than X_i. The instance I is satisfied if there is an assignment of one colour to each variable that satisfies every constraint.

3 The Algorithm

The algorithm $LowdegSAT(F)$ for determining the satisfiability of a CNF formula F is shown in Figure 1. It is shown as a list of cases, where the earliest case that applies is used, *e.g.* case 8 is only used if none of cases 0–7 apply. Case 0 is a base case. Cases 1–5 are referred to as *simple reductions*, since the effect of these cases is only to remove literals or variables from F, without adding any new literals or variables. Cases 6–7 are the non-simple reductions, and cases 8–12 are branchings.

We use $f(F) = L(F) - 2N(F) = \sum_{v \in Vars(F)}(d(v,F) - 2)$ as a measure of complexity, motivated by the fact that this is the maximum number of occurrences of v that need to be removed from F before v can be removed by a polynomial-time reduction. As an indication that this is a relevant measure, when using this measure the worst cases for $d(v) = 3$ and $d(v) = 4$ both get the same branching number, as we see in the next section. $f(F)$ is also used in the algorithm for deciding whether to apply certain reductions and branchings or not.

We say that a formula F' is the *step k-reduced version* of F if F' is the result of applying the reductions in cases 0–k, in the order in which they are listed, until no such reduction applies anymore. F' is called step k-reduced (without reference to F) if no reduction in case k or earlier applies (*i.e.* F is step k-reduced if $LowdegSAT(F)$ reaches case $k+1$ without applying any reduction). A synonym to step 7-reduced is fully reduced.

Definition 1. Standardising *a CNF formula F refers to applying the following reductions as far as possible:*

1. *Subsumption: If there are two clauses C, D in F, and if every literal in C also occurs in D, then D is* subsumed *by C. Remove D from F.*

Algorithm LowdegSAT(F)

Case 0: If $F = \emptyset$, return 1. If $\emptyset \in F$, return 0.

Case 1: If F is not on standard form, standardise it (see Def. 1).

Case 2: If there is some 1-clause $(l) \in F$, return $LowdegSAT(F[l])$.

Case 3: If there is a pure literal l in F, return $LowdegSAT(F[l])$.

Case 4: If there is a 2-clause $(l_1 \vee l_2)$ and a clause $D = (l_1 \vee \neg l_2 \vee C)$ in F for some possibly empty C, construct F' from F by deleting $\neg l_2$ from D and return $LowdegSAT(F')$.

Case 5: If there is a variable x in F with at most one non-trivial resolvent (such as a $(1, 1)$-variable), return $LowdegSAT(DP_x(F))$.

Case 6: If there is a variable x in F with $d(x) = 3$ such that resolution on x is admissible (see Def. 2), return $LowdegSAT(DP_x(F))$.

Case 7: If there are two clauses $(C \vee D), (C \vee E)$, with $|C| > 1$, construct F' from F by replacing these two clauses by $(C \vee \neg x), (x \vee D), (x \vee E)$ for a newly introduced variable x, and return $LowdegSAT(F')$.

Case 8: If $d(F) > 3$, pick a variable x of maximum degree. If some literal of x, assume $\neg x$, occurs in a single clause $(\neg x \vee l_1 \vee \ldots \vee l_k)$, return $LowdegSAT(F[x]) \vee LowdegSAT(F[\{\neg x, \neg l_1, \ldots, \neg l_k\}])$. If both x and $\neg x$ occur in at least two clauses, return $LowdegSAT(F[x]) \vee LowdegSAT(F[\neg x])$.

Case 9: If there is a 2-literal l such that $f(F)$ reduces by at least six in a branch $F[l]$, assume that $\neg l$ occurs in a clause C along with literals $l_1, \ldots, l_k$ and return $LowdegSAT(F[l]) \vee LowdegSAT(F[\{\neg l, \neg l_1, \ldots, \neg l_k\}])$.

Case 10: If there is a clause $C = (\neg v_1 \vee \ldots \vee \neg v_k)$ that contains only 1-literals, and $|C| \geq 4$, return $LowdegSAT(F - C + (\neg v_1 \vee \ldots \vee \neg v_{\lfloor k/2 \rfloor})) \vee LowdegSAT(F - C + (\neg v_{\lfloor k/2 \rfloor + 1} \vee \ldots \vee \neg v_k))$.

Case 11: Let a be a 2-literal (assumed to be positive) with a maximum number of neighbours. Let the clause that contains $\neg a$ be $(\neg a \vee \neg b \vee \neg c)$. If the literal a has at least three neighbours, return $LowdegSAT(F[a]) \vee LowdegSAT(F[\{\neg a, b, c\}])$.

Case 12: If no previous case applied, the formula can be converted into a $(3, 2)$-CSP instance with $N(F)/3$ variables, as described in Lemma 1. Perform this conversion, and apply Eppstein's algorithm from [8].

Fig. 1. Algorithm for SAT when most variables have few occurrences

2. *Trivial clauses: If there is a clause C in F such that both literals v and $\neg v$ occur in C for some variable v, then C is a trivial clause. Remove it from F.*
3. *Multi-occurring literals: If there is a clause C in F where some literal l occurs more than once, remove all but one of the occurrences of l from C.*

A formula F where none of these reductions apply is said to be on standard form.

Definition 2. *Let F be a step 5-reduced SAT formula, and let F' be the step 5-reduced version of $DP_x(F)$, for some variable x that occurs in F. Then, resolution on x in F is* admissible *if $f(F') \leq f(F)$,* i.e. *$L(F) - L(F') \geq 2(N(F) - N(F'))$.*

Definition 3. Backwards resolution *is the operation of replacing two clauses $(C \vee D)$, $(C \vee E)$ in a formula F with $(\neg a \vee C), (a \vee D), (a \vee E)$ for a new* *variable a. To avoid loops of reductions, we only use this when* $|C| > 1$*, so that the net difference in* $f(F)$ *is strictly positive.*

Lemma 1. *Given a 3-regular SAT formula* F *where all 2-literals occur only in 2-clauses and all 1-literals occur only in 3-clauses, there is a corresponding* $(3,2)$*-CSP instance* I*, constructible in polynomial time and with* $N(F)/3$ *variables, that is satisfiable if and only if* F *is satisfiable.*

Proof. By Lemma 14.6 of [13], a formula F with c 3-clauses and otherwise only 2-clauses can be converted into an instance I of $(3,2)$-CSP with c variables (by first creating one variable in I for each clause in F, and then performing a reduction used by Eppstein in [8] to remove every variable with only two values). Since every variable of F occurs in only one 3-clause, the resulting instance I has $N(F)/3$ variables. □

Next we show the correctness of *LowdegSAT*. First we show the correctness of the branching that is used in some of the cases (introduced in [12], under the name complement search), and then the overall correctness of the algorithm. The proof contains some references to lemmas in the next section, as we feel that the correctness belongs to this section while these other lemmas are more easily shown in the context of algorithm analysis. No circular references occur, since no lemma in the analysis section refers to this lemma.

Lemma 2. *Let* $\neg x$ *be a 1-literal in a formula* F*, and let the clause where* $\neg x$ *occurs be* $C = (\neg x \vee l_1 \vee \ldots \vee l_d)$*. Then either* $F[x]$ *is satisfiable, or* $F[\neg x]$ *and* $F[\{\neg x, \neg l_1, \ldots, \neg l_d\}]$ *are equi-satisfiable (*i.e. *either both are satisfiable, or neither).*

Proof. Assume that $F[x]$ is unsatisfiable. Then, if there is a satisfying assignment A to F, it must set $\neg x$ to true and changing the value of x in A must create an unsatisfied clause. The only possible such clause is C, which means that all other literals of C must be false in A. □

Lemma 3. *The algorithm LowdegSAT applied to a CNF formula* F *correctly calculates the satisfiability of* F*.*

Proof. Case 0 is correct by the definition of the problem, and cases 1–4 are easily checked. Cases 5–7 use resolution, and the correctness of this operation is proven in *e.g.* [7]. Furthermore, the reduction process will terminate. If F' is the step 3-reduced version of F, by Lemma 4 the simple reductions keep $f(F')$ non-increasing and decrease $L(F')$. Resolution keeps $f(F)$ non-increasing while decreasing the number of variables, implying that $L(F)$ decreases, and creates no singletons. Backwards resolution decreases $f(F)$ strictly and also creates no singletons. This shows that no infinite chain of reductions is possible from F' onwards, and the process of applying reductions 1–3 is certainly finite. Cases 8,9 and 11 either use a branching with two assignments x and $\neg x$, which is obviously correct, or branchings that are correct by Lemma 2. Case 10 is correct, as any assignment that satisfies C must satisfy at least one of the new clauses. In case 11, the length of the clause containing the 1-literal must be 3, as a 2-clause with a 1-literal $\neg x$ implies that resolution on x is admissible (see Lemma 6). The correctness and completeness of case 12 given that cases 0–11 do not apply is proven in Lemma 15 in the next section, as this proof uses a number of other lemmas, that are best shown in the context of the algorithm analysis. □

4 Analysing the Running Time

This section contains the proof of the upper bound $O(1.1279^{f(F)})$ on the worst-case running time of the algorithm, presented by lemmas roughly following the cases of the algorithm. The section is split into subsections in the following way: Section 4.1 presents the method of analysis and contains some results regarding this and the measure $f(F)$. Section 4.2 deals with cases 0–7 of the algorithm, and gives the basic structural properties that are enforced there. Section 4.3 deals with case 8, where all variables with degrees higher than 3 are handled, and Section 4.4 with case 9, where most of the structure of the problem is enforced. Finally, Section 4.5 deals with the rest of the cases and concludes the proof of the running time of the algorithm. With the help of the structure enforced by case 9, cases 10-11 are easier cases that prepare for the applicability of the CSP construction in case 12.

4.1 Technical Aspects of the Method of Analysis

We use Kullmann's method from [13] to get an upper limit on the running time of *LowdegSAT*. In summary, if F is a fully reduced formula and if *LowdegSAT* applied to F branches into formulas $F[A_1], \ldots, F[A_d]$, let F_i be the fully reduced version of $F[A_i]$ for $i = 1, \ldots, d$. The branching number of this particular branching is $\tau(f(F) - f(F_1), \ldots, f(F) - f(F_d))$, where $\tau(t_1, \ldots, t_d)$ is the unique positive root to the equation $\sum_i x^{-t_i} = 1$. If α is the biggest branching number for all branchings that can occur in *LowdegSAT*, then the running time of the algorithm for a formula F is $O(\alpha^{f(F)})$. For a more detailed explanation, see Kullmann's paper.

As mentioned, we use $f(F) = L(F) - 2N(F) = \sum_{v \in Vars(F)}(d(v, F) - 2)$ as a measure of complexity. While this measure might seem odd at first, there is an intuitive reading of it: $d(v, F) - 2$ is the number of occurrences of v that need to be removed before a simple reduction on v is possible. Thus, it can be viewed as assigning a weight to each variable depending on its degree, so that one variable of degree four counts for as much as two variables of degree three. If F is 3-regular (such as after case 8 of the algorithm), $f(F) = N(F)$, but even for a 3-regular F we use $f(F)$ to guide when we should apply certain reductions and branchings. Furthermore, $f(F)$ obeys all the required properties of a measure, provided that F is free of singletons, as proven in the next lemma.

Lemma 4. *Let F be a CNF formula with $d(x) \geq 2$ for every variable x occurring in F. Let F' be the fully reduced version of F. Then, $f(F) \geq 0$, $f(F) = 0$ if and only if F' is empty, and $f(F') \leq f(F)$.*

Proof. The first is obvious from $f(F) = \sum_{v \in Vars(F)}(d(v,F)-2)$. For the second, note that the simple reductions never increase the degree of any variable, and that no variable of degree 2 can remain after the simple reductions have been applied. $f(F) = 0$ if and only if all variables in F have degree 2, which implies that all variables will be removed by the simple reductions.

For the final part, $f(F)$ is non-increasing over all simple reductions, by the previous observations, and over standard and backwards resolution whenever these are applied, by the restrictions in the algorithm. To complete the argument, we only need to show that no singletons are created in these steps. For standard resolution, a variable that occurs in only one resolvent must co-occur exactly once with x in F, as both resolvents are non-trivial. For backwards resolution, the only change in the degrees of variables is that the variables in C have their degrees decreased by one. Thus, $f(F)$ is non-increasing over the entire process of reductions. □

Given these properties, we need one more lemma to give a lower bound for $f(F) - f(F')$ when F' is the result of an assignment and reductions on F.

Lemma 5. *Let F be a fully reduced formula, A an assignment to variables of F and F' the reduced version of $F[A]$. Further, let F_0 be the result of a sequence of applications of the reductions in cases 1–3 in any order to $F[A]$. If F_0 is free of singletons, and if F' contains no empty clause, we have $f(F') \leq f(F_0)$.*

Proof. This can be shown by induction on the number of reductions applied. Remember that cases 1–3 only remove clauses and literals from F, and note that the only case of these that will ever remove the last occurrence of a literal from a clause without also removing the entire clause is case 2.

First, if some reduction is applicable on $F[A]$, then every clause and literal that would be removed by the application of this reduction will be removed by any sequence of applications of cases 1–3 ending in a step 3-reduced formula. This can be verified without any great difficulty (using the above observations and the fact that F' contains no empty clause).

Second, assume that the induction hypothesis is true for every sequence of k of these reductions acting on $F[A]$; that is, for any sequence of k applications of cases 1–3 acting on $F[A]$, removing a set of clauses C^* and a set of literals L, every possible sequence of such reductions ending in a step 3-reduced formula will remove at least these clauses and literals. It can be verified without any great difficulty that any extra clauses and literals that would be removed by the application of one further reduction will also be missing in any resulting step 3-reduced formula (again using that F' contains no empty clause).

Thus, if F_1 is the true step 3-reduced version of $F[A]$, then for every variable v that occurs in both F_0 and F_1, $d(v, F_0) \geq d(v, F_1)$. If F_0 is free of singletons, this gives us $f(F_0) \geq f(F_1)$, and the previous lemma gives us $f(F_1) \geq f(F')$. □

Thus, even though we do not know the exact sequence of reductions that will be made after a particular assignment (this is not even defined in the algorithm), this result guarantees that we can safely underestimate the amount of reduction due to cases 1–3. We will permit ourselves to say that a particular chain of reductions occurs, rather than using cumbersome more exact phrases. Calculating $f(F) - f(F_0)$ is easily done using $f(F) = \sum_v (d(v, F) - 2)$.

4.2 Basic Structural Properties

This section contains some results regarding the basic structural properties that exist in a fully reduced formula. First we give a lemma that shows a sufficient condition for when resolution on a variable x is admissible.

Lemma 6. *Let F be a step 5-reduced CNF formula, and x, $d(x) = 3$, be a variable occurring in F such that applying resolution to x increases the degree of at most c variables, while the resolution together with the reductions in cases 1–5 removes or decreases the degree of at least c variables, including x. Then resolution on x is admissible.*

Proof. Use $f(F) - f(F') = \sum_{v \in Vars(F)} (d(v, F) - 2) - \sum_{v \in Vars(F')} (d(v, F') - 2)$, where F' is the step 5-reduced version of $DP_x(F)$. Note that since $d(x, F) = 3$, a variable can increase its degree by at most one in the resolution process. □

Next, Lemmas 7–8 show the mentioned structural properties.

Lemma 7. *If F is a 3-regular, fully reduced formula, and if C, D are two clauses in F, then $|Vars(C) \cap Vars(D)| \leq 2$. If $|C| = 2$, then $|Vars(C) \cap Vars(D)| \leq 1$, so $Vars(C) \not\subseteq Vars(D)$.*

Proof. For the first part, note that some reduction applies both if $l_1, l_2 \in C$ and $l_1, l_2 \in D$, and if $l_1, l_2 \in C, \neg l_1, \neg l_2 \in D$. There is no way for C and D to share three variables without one of these cases occurring. For the second part, if $C = (l_1 \vee l_2)$ and $l_1, \neg l_2 \in D$, then case 4 applies and D is shortened. □

Lemma 8. *Let F be a step 5-reduced formula, and let a, b be $(2, 1)$-variables in F. The following structures all guarantee an admissible resolution.*

1. *2-clause C with $\neg a \in C$*
2. *3-clause C with $\neg a, b \in C$ and clause D with $a, b \in D$*
3. *3-clause C with $\neg a, l \in C$, clause D with $a, b \in D$ and 2-clause $(\neg l \vee b)$ for some literal l.*

Proof. In the first two cases, we see immediately by Lemma 6 that resolution on a is admissible. In the third case, we see that one resolvent is either a copy of an existing clause or will be shortened or removed in case 4 at the latest. In either case, $f(F)$ has increased by at most 1 in the resolution process, and at least one simple reduction which strictly decreases $f(F)$ applies, guaranteeing that resolution on a is admissible. □

With these tools, we can prove that all cases 8–12 get a branching number of $\tau(4, 8)$ or better.

4.3 Case 8: Variables of Higher Degree

Here, we prove that the branching number is sufficiently good when branching on any variable x with $d(x) > 3$.

Lemma 9. *If F is a reduced formula with $d(F) > 3$, then applying case 8 of the algorithm results in a branching number of at most $\tau(4, 8)$.*

Proof. For any variable y occurring in any 2-clause with x, we are limited to the following options: $d(y) = 3$ so that the 2-literal y occurs in a 2-clause and x and y have no co-occurrences in any other clauses; $d(y) > 3$ and 2-clauses $(x \vee y), (\neg x \vee \neg y)$ are the only co-occurrences of x and y; or finally $d(y) > 3$ and x and y co-occur in only one 2-clause, say $(x \vee y)$. In the latter case, one longer clause $(\neg x \vee \neg y \vee C)$ for some C can occur. Similarly, for any variable y occurring with x, but not in any 2-clause, we have the following options: $d(y) = 3$ and x and y co-occur only once; $d(y) = 3$ and x and y co-occur only in clauses $(x \vee y \vee C), (x \vee \neg y \vee D)$ (or similarly with $\neg x$), where C and D do not share variables and $|D| > 1$ if $\neg y$ is a 1-literal; or finally $d(y) > 3$, where x and y can co-occur several times as long as the same pair of literals never occurs in more than one clause (in other words, the variable y occurs at most twice with the literal x). From all of this, we can infer the following: The reduction in $f(F)$ in a branch $F[x]$ is at least $d(x) - 2$ plus the number of 2-clauses that contain the variable x plus the contribution from the longer clauses involving the literal x. With only one such clause, this contribution is at least 2. With two such clauses, the contribution is at least 4. With three, the contribution is at least 5, occurring in a situation such as clauses $(x \vee y \vee a), (x \vee z \vee b), (x \vee \neg y \vee \neg z \vee c)$, if $d(v) = 3$ for every involved variable v. Let a be the reduction of $f(F)$ in the x branch, and b the reduction in the $\neg x$ branch. If each literal x and $\neg x$ is involved with at most two clauses longer than a 2-clause, then $a + b \geq 12$ and we need only prove that $a, b \geq 4$. Assume that $\neg x$ has at most as many occurrences as x.

If $\neg x$ is at least a 2-literal, or a 1-literal present in a 3-clause or longer clause, then the result is immediate. If $\neg x$ is a 1-literal present in a 2-clause, say $(\neg x \vee y)$, then the extra assignment $\neg y$ will ensure the result.

The remaining case is that there are three longer clauses containing the literal x, which ensures a reduction of $f(F)$ in the x branch of at least 7 plus the contribution from 2-clauses. If $\neg x$ is at least a 2-literal, then either there are two 2-clauses and a reduction of at least 4 in the $\neg x$ branch, or a reduction of at least 5 in the $\neg x$ branch. If $\neg x$ is a 1-literal, finally, then we shall see that $f(F)$ reduces by at least 5 in the branch $\neg x$. If $\neg x$ occurs in a 3-clause or longer, or in a 2-clause $(\neg x \vee y)$ where $d(y) > 3$, then the immediate assignments reduce $f(F)$ by at least 4 and at least one more variable is affected by the assignments. If $\neg x$ occurs in a 2-clause $(\neg x \vee y)$ with $d(y) = 3$, then y must be a 2-literal, so that $\neg y$ is a 1-literal occurring in a clause of length at least 3. This concludes the proof. □

In every case after this one, F is 3-regular.

4.4 Case 9: Imposing More Structure

We give some conditions under which case 9 of the algorithm applies, and show that the branching number will be at most $\tau(6,6)$.

Lemma 10. *If F is a 3-regular, fully reduced formula, the following statements are true. For the sake of convenience, assume w.l.o.g. that for any variable v, the literal $\neg v$ occurs only once in F.*

1. *Any branch $F[\neg a]$ for a variable a reduces $f(F)$ by at least 6.*
2. *Any branch $F[a]$ for a variable a where the literal a occurs in some clause C with $|C| \geq 5$ reduces $f(F)$ by at least 6.*
3. *If literals a, b occur together in one clause, and $a, \neg b$ occur together in another, then a branch $F[a]$ reduces $f(F)$ by at least 6.*

Proof. 1: Let S be the set of literals that occur in a clause together with $\neg a$ in F. For every literal $l \in S$, $\neg l$ is assigned in the branch. We know that if $l_1, l_2 \in S$, then any clause containing $\neg l_1$ does not contain $\neg l_2$ or a, and a clause C with $\neg l_1, l_2 \in C$ has $|C| > 2$ and $|S| > 2$ if l_1 is a negated literal, $|C| > 3$ if l_1 is an unnegated literal. Either way, each assignment $\neg l_i$ affects at least two literals not from the variables in S.

If $|S| \geq 3$, then at least four variables are assigned in the branch, and at least six literals beyond these are removed from F. By a simple counting argument, this requires at least six variables to be affected.

If $|S| = 2$, let $S = \{l_1, l_2\}$ where l_1, l_2 are some literals for variables b and c, respectively.

If some clause C contains both literal $\neg l_1$ and variable c, then by necessity $l_1 = b$, $l_2 = c$ and $C = (\neg b \vee c \vee C')$ where $|C'| \geq 2$ and C' contains no literals of variables a, b, c. In this case, no clause containing $\neg c$ can be formed without using a sixth variable, by Lemma 7.

Otherwise, any clause containing $\neg l_i$ for $i = 1, 2$ has no other variable in common with the clause containing $\neg a$. We have three further cases, depending on the negations in S.

If $S = \{b, c\}$, then there must exist clauses $(\neg b \vee C), (\neg c \vee D)$ with $|C|, |D| \geq 2$. If less than six variables are affected, $Vars(C) = Vars(D)$ and $|C| = |D| = 2$, but then, either resolution or backwards resolution is admissible on a variable in C. Otherwise, at least six variables are removed in the branch.

If $S = \{b, \neg c\}$, then there exist clauses $(\neg b \vee C)$ with $|C| \geq 2$ and $(c \vee D)$, $(c \vee E)$ with $|D|, |E| \geq 1$. If less than six variables are affected, $|D| = |E| = 1$ and $Vars(C) = Vars(D) \cup Vars(E)$, and by Lemma 8, we must have clauses $(\neg b \vee \neg u \vee \neg v), (c \vee u), (c \vee v)$ for variables u, v. Now, the second appearances of literals u and v must occur in different clauses, where no other literal of the variables a, b, c, u or v can occur. Counting these clauses, at least six variables are removed in the branch.

If $S = \{\neg b, \neg c\}$, then we have clauses $(b \vee A), (b \vee B), (c \vee C), (c \vee D)$, where no case uses only six variables. By Lemma 8 and since case 7 does not apply, we have $A, B \neq C, D$, and by Lemma 7, $Vars(A) \neq Vars(B)$ and $Vars(C) \neq Vars(D)$, so either A–D are all of length one with distinct variables (for a reduction of at least 7 in the branch) or at least one, say C, has $|C| > 1$. In the latter case, D still introduces a variable not in C, for a total reduction of at least 6.

2: Assume w.l.o.g. that $C = (a \vee l_1 \vee l_2 \vee l_3 \vee l_4)$, where l_1–l_4 are literals of variables b–e, respectively. By assumption, there is one more clause D containing literal a, and by Lemma 7, D contains at least one variable other than a–e. At least six variables are affected by the assignment a.

3: By Lemma 8, the clauses can w.l.o.g. be assumed to be $(a \vee b \vee l_1)$, $(a \vee \neg b \vee l_2 \vee l_3)$ where l_1–l_3 are literals of variables c–e. If the reduction in $f(F)$ is less than 6, the second occurrence of literal b must occur in a clause using only these variables. No such clause can exist. □

Lemma 11. *Let F be a 3-regular, fully reduced SAT formula where no condition from Lemma 10 applies. Assume w.l.o.g. that for every variable v, literal $\neg v$ is a 1-literal. Then the following statements hold:*

1. *If there is a clause C with literals a, $\neg b$ and $\neg c$ for some variables a, b, c, then a branch $F[a]$ reduces $f(F)$ by at least 6.*
2. *If there is no such clause, but there is a clause C with literals a and $\neg b$ for some variables a, b, then a branch $F[a]$ reduces $f(F)$ by at least 6.*

Proof. 1: Assignments b and c will be made. By the various restrictions on F, it can be verified that whether $C = (a \vee \neg b \vee \neg c)$ or $C = (a \vee \neg b \vee \neg c \vee l_1)$, at least six variables are affected.

2: Assignment b will be made, and $|C| \geq 3$. By similar arguments as before, considering both clauses that contain the literal a as well as the clauses containing literal b, at least six variables are affected. □

We see that for any F where none of cases 0–9 apply, we have a specific structure where every clause C contains either only 2-literals, in which case $2 \leq |C| \leq 4$, or only 1-literals, in which case $|C| \geq 3$. Additionally, every pair of variables co-occurs in at most one clause.

4.5 The Final Cases

Given the structure imposed by case 9, showing the rest of the results is relatively easy. Case 10 imposes a stricter limit on the length of a clause with 1-literals, case 11 gives us stronger guarantees on the neighbourhood of a 2-literal, and finally, if all other cases fail to apply, case 12 can be applied to convert the formula to an instance of $(3, 2)$-CSP.

Lemma 12. *Let F be a SAT formula where case 10 is the earliest case of the algorithm LowdegSAT that applies. The branching number for this case is at least $\tau(6, 6)$.*

Proof. Let C be the clause that is being split. For any literal $l_i \in C$ that is not included in the new clause, $\neg l_i$ becomes a pure literal, so that an assignment $\neg l_i$

is made. For each such assignment, two literals for other variables are affected. If there are at least three such assignments $\neg l_i$, we have at least six additional literals, and by a counting argument at least six variables are affected in total.

If only two literals become pure, say a, b, let S_i for $i = 1, 2$ be the set of literals v such that v occurs in i clauses together with literal a or b. Assume w.l.o.g. that $S_1 = \{u_1, \ldots, u_d\}$ and $S_2 = \{v_1, \ldots, v_e\}$. $|S_1| + 2|S_2| \geq 4$, and for every literal $l \in S_2$ an additional assignment $\neg l$ is made. We trace these assignments.

First, if $S_2 = \emptyset$, the reduction in $f(F)$ is at least $2 + |S_1| \geq 6$.

Second, if $|S_2| = 1$, $|S_1| \geq 2$. Let D be the clause where $\neg v_1$ occurs. If less than six variables are to be removed, $D = (\neg u_1 \vee \neg u_2 \vee \neg v_1)$, but then u_1 and u_2 are assigned and must lie in different clauses, which requires extra variables.

Third, if $|S_2| = 2$, then some literal $\neg w$ shares a clause with some $\neg v_i$, and assignment w is made. At least one occurrence of w is in a clause with some new variable, for a reduction of at least 6.

Finally, $|S_2| \geq 3$. If $|S_2|+|S_1| > 3$, then the reduction is at least 6. Otherwise, some extra variable is required to form a clause with $\neg v_i$. □

Lemma 13. *Let F be a CNF formula such that no case before case 11 of LowdegSAT applies. Let a be a variable. W.l.o.g., assume that literal $\neg a$ occurs once in F. Then, if a is a member of k 2-clauses, an assignment $F[\neg a]$ reduces $f(F)$ by at least $7 + k$.*

Proof. Let the clause that contains $\neg a$ be $(\neg a \vee \neg b \vee \neg c)$, so that assignments b and c are made. Each 2-clause containing a, b or c contributes one variable, and if there are l literals otherwise removed from F, these literals belong to at least $l/2$ variables. Since no clause contains both b and c, the result follows from this. □

Lemma 14. *If F is a CNF formula such that case 11 is the earliest case of LowdegSAT that applies, then the branching number is at most $\tau(4, 8)$. If case 11 does not apply either, then every 2-literal l is involved in two 2-clauses.*

Proof. If a is part of no 2-clauses, then the number of variables affected by assignment a is at least 5, which by Lemma 13 leads to a branching with a branching number of at most $\tau(5, 7) < \tau(4, 8)$. If literal a is neighbour to only three other variables, a must be involved in one 2-clause, and by the same lemma, we have a branching with a branching number of at most $\tau(4, 8)$. The remaining case, with only two other variables, can only be achieved by two 2-clauses. □

Lemma 15. *If F is a CNF formula such that no case among cases 0–11 of LowdegSAT applies to F, then the construction in Lemma 1 is applicable, and the total time for LowdegSAT(F) is $O(1.3645^{N(F)/3}) \subset O(1.1092^{f(F)})$.*

Proof. In addition to the structural properties noted previously, we have by case 10 that $|C| = 3$ for every clause C with 1-literals and by case 11, as noted in Lemma 14, $|C| = 2$ for every clause C with 2-literals, which proves the applicability of the construction. Eppstein's algorithm [8] runs in time $O(1.3645^n)$, and the resulting CSP instance has $N(F)/3$ variables. With $f(F) = N(F)$ at this point in the algorithm, we get the described running time. □

This concludes our sequence of lemmas. We will proceed with the main theorem.

Theorem 1. *If F is a CNF formula without singletons, then LowdegSAT(F) decides the satisfiability of F in time $O(1.1279^{L(F)-2N(F)})$.*

Proof. This follows from the various relevant lemmas. □

Corollary 1. *If F is any CNF formula where the degree of a variable x is limited to at most d, the running time is in $O(1.1279^{(d-2)\cdot N(F)})$.*

Proof. All singletons will be removed in simple reductions before any branching is done. If F' is the step 3-reduced version of F, then $d(v, F') \leq d$ for every $v \in Vars(F')$ and $L(F') - 2N(F') \leq (d-2) \cdot N(F') \leq (d-2) \cdot N(F)$. □

5 Conclusions

We have presented an algorithm which decides the satisfiability of a CNF formula F, with $N(F)$ variables and of length $L(F)$, in time $O(1.1279^{L(F)-2N(F)})$ if F is free of singletons. This implies a running time of $O(1.1279^{(d-2)N(F)})$ when F is either a formula with at most d occurrences for any variable or a singleton-free formula with d occurrences per variable on average. For $d \leq 4$, this is better than previous results.

No previous algorithms have been published for SAT problem instances where the number of occurrences per variable is limited. Looking at other NP-hard problems, we find only a few papers where similar attacks have been made. In [2], Chen et al. apply advanced methods of algorithm analysis to get an algorithm that solves the Vertex Cover problem for a graph of maximum degree 3 in parameterised time $O(1.194^k k + n)$ for a maximum cover size of k, and in time $O(1.1255^n)$ for the non-parameterised version. Closer to the work in this paper, one of the helper algorithms for the problem of counting max-weight models for 2SAT formulae in [3] is an algorithm that runs in time $O(1.1892^{N(F)})$ when $d = 3$. This helper algorithm is then used to extend this into running times of $O(1.2400^{N(F)})$ when $d = 4$ and $O(1.2561^{N(F)})$ for the general problem.

For future research, it could be fruitful to apply techniques similar to or inspired by those in [3] to extend the results in this paper to an algorithm that is more effective when $d > 4$.

References

[1] Tobias Brueggemann and Walter Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1–3):303–313, 2004.

[2] Jianer Chen, Iyad A. Kanj, and Ge Xia. Labeled search trees and amortized analysis: Improved upper bounds for NP-hard problems. In *Proceedings of the 14th Annual International Symposium on Algorithms and Computation (ISAAC 2003)*, pages 148–157, 2003.

[3] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting models for 2SAT and 3SAT formulae. *Theoretical Computer Science*, 332(1-3):265–291, 2005.

[4] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon M. Kleinberg, Christos H. Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.

[5] Evgeny Dantsin, Edward A. Hirsch, and Alexander Wolpert. Algorithms for SAT based on search in Hamming balls. In *Proceedings of the 21st Annual Symposium on Theoretical Aspects of Computer Science (STACS 2004)*, volume 2996 of *Lecture Notes in Computer Science*, pages 141–151. Springer, 2004.

[6] Evgeny Dantsin and Alexander Wolpert. Derandomization of Schuler's algorithm for SAT. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 69–75, 2004.

[7] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[8] David Eppstein. Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms (SODA 2001)*, pages 329–337, 2001.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[10] Edward A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.

[11] Kazuo Iwama and Suguru Tamaki. Improved upper bounds for 3-SAT. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, page 328, 2004.

[12] Paul Walton Purdom Jr. Solving satisfiability with less searching. In *IEEE Transactions on Pattern Analysis and Machine Intelligence. PAMI-6*, pages 510–513, jul 1984.

[13] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223:1–72, 1999.

[14] Uwe Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002.

[15] Rainer Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms*, 54(1):40–44, 2004.

[16] Stefan Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. In *Proceedings of the 9th Annual International Conference on Computing and Combinatorics (COCOON 2003)*, pages 548–558, 2003.

[17] Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8:85–89, 1984.

A New Approach to Model Counting

Wei Wei and Bart Selman

Department of Computer Science,
Cornell University,
Ithaca, NY 14853

Abstract. We introduce ApproxCount, an algorithm that approximates the number of satisfying assignments or models of a formula in propositional logic. Many AI tasks, such as calculating degree of belief and reasoning in Bayesian networks, are computationally equivalent to model counting. It has been shown that model counting in even the most restrictive logics, such as Horn logic, monotone CNF and 2CNF, is intractable in the worst-case. Moreover, even approximate model counting remains a worst-case intractable problem. So far, most practical model counting algorithms are based on backtrack style algorithms such as the DPLL procedure. These algorithms typically yield exact counts but are limited to relatively small formulas. Our ApproxCount algorithm is based on SampleSat, a new algorithm that samples from the solution space of a propositional logic formula near-uniformly. We provide experimental results for formulas from a variety of domains. The algorithm produces good estimates for formulas much larger than those that can be handled by existing algorithms.

1 Introduction

In recent years, we have seen tremendous improvements in our abilities to solve large Satisfiability (SAT) problems. The field benefits from significant progress both in terms of our theoretical understanding of the SAT problem, and in terms of practical algorithm design and engineering of solvers. Because the state-of-the-art SAT solvers can often handle formulas with thousands of variables [19], problems from many other domains, such as planning [8] and microprocessor verification [16], have been encoded to SAT instances, and solved effectively by SAT solvers. This indirect approach is often competitive with, if not faster than, solving the problems directly with methods developed specifically for the domain under consideration.

Broadly speaking the "SAT approach" appears to work quite well for NP-complete problems. However, many tasks in AI belong to higher complexity classes than NP, and therefore, presumably, cannot be encoded as SAT problems, without introducing exponentially many variables. A number of these tasks are computationally equivalent to counting the satisfying assignments of a formula in propositional logic [11]. For example, consider a knowledge base (KB) written in propositional logic with no explicit probabilistic information and a statement

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 324–339, 2005.

s. The degree of belief in s, which is defined as $P(s|KB)$, can be calculated by $\frac{\mathcal{M}(s \wedge KB)}{\mathcal{M}(KB)}$, where $\mathcal{M}(\cdot)$ is the model (satisfying assignment) count of the input formula. Moreover, probabilistic reasoning using Bayesian belief networks can also to be reduced effectively to counting models of Boolean formulas.

It has been shown that many algorithms and techniques developed for SAT, such as DPLL [4] and clause learning [9], can be adapted to solve model counting problems. However the memory usage and run time of such algorithms often increase exponentially with problem size, and such algorithms are therefore limited to relatively small formulas.

In this paper, we present ApproxCount, an approximate model counting algorithm based on a biased random walk strategy. Random walk strategies have been shown to be effective on a variety of SAT instances [13]. Recently, a biased random walk strategy was used in SampleSat [17], a new algorithm that samples from the solution space of a Boolean formula near-uniformly. Following the scheme outlined by Jerrum *et al* [7], ApproxCount uses SampleSat to repetitively draw samples from the solution spaces of a Boolean formula and its sub-formulas.

A key advantage of a sampling approach for counting over approaches based on complete backtrack search is that the run time can be controlled much better. After the first several samples (satisfying assignments) are drawn, we can obtain a reasonably accurate estimate of total run time depending on how accurate we want our final count to be. In applications where a decision has to be made by a certain deadline, the algorithm can adapt by drawing fewer samples in each iteration.

The number of calls ApproxCount makes to SampleSat is determined by the number of variables in the formula under consideration. An intriguing question is whether the error in each intermediate result due to SampleSat's deviation from uniformly sampling will accumulate to an unmanageably large overall error rate. After all, even a small error in each step can potentially lead to an exponential deviation from the exact count, which would make our approach of little practical use. Somewhat surprisingly, our experiments show that the errors from setting different variables appear to offset each other, resulting in an overall final estimate that is within a factor two of the exact count in many domains, including random 3CNF formulas, several structured problems, and constructed combinatorial instances.

For random 3CNF formulas, the complexity of model counting is determined by the clause/variable (C/V) ratio. Low C/V ratios are particularly difficult for counting procedures because of the very large number of satisfying assignments. In recent years, in work on DPLL based model counters, the C/V ratio that requires the largest run time has shifted from 1.2 to 2.0, with the introduction of new techniques and algorithms. In our work, we show that the error rate of ApproxCount on random 3CNF formulas stays relatively constant for different C/V ratios, when given the same total run time. This suggests that our approach is more robust compared to the DPLL style techniques.

The difficulty in evaluating our ApproxCount algorithm is that exact model counting programs often do not work on larger instances. We therefore introduce a class of structured formulas, where we know the exact number of satisfying assignments (obtained from basic combinatorics). For DPLL style methods, the run time explodes exponentially with the problem size. However, we will show that ApproxCount approximates the model counts will high accuracy, while the run time scales only polynomially with problem size.

The reminder of this paper is organized as follows. Background with theoretical results and development of existing model counting approaches is discussed in the next section. In Section 3, we review the SampleSat algorithm. Section 4 provides a detailed description of our new approximate model counting algorithm, and discusses on several implementation issues. Experimental results are given in Section 5. We then discuss how to extend our algorithm to deal with real numbers in probabilistic reasoning. In the last section, we summarize the main results of this paper.

2 Background

Before we start our discussion, we first define some terms that we use throughout this paper. Without loss of generality, the Boolean formulas we discuss are in their conjunction normal form (CNF) unless otherwise indicated. A CNF is a conjunction of clauses. A clause is defined as a disjunction of literals. A literal is a Boolean variable, which ranges over $\{\mathit{True}, \mathit{False}\}$, or its negation. When an assignment of truth values to the Boolean variables makes a formula evaluate to *True*, the formula is said to be satisfied, and the assignment is called a satisfying assignment, or interchangeably, a model of the formula. $\mathcal{M}(F)$ denotes the number of unique satisfying assignments of formula F.

2.1 Complexity Results

The problem of counting the satisfying assignments of a propositional logic formula is #P-complete [15], a complexity class that is at least as hard as the polynomial-time hierarchy [14]. To approximate the model count within δ, we look for a number M that satisfies

$$M/(1+\delta) \leq \mathcal{M}(F) \leq M(1+\delta).$$

For any positive δ, approximating the model count of an arbitrary Boolean formula F is NP-hard. This is straightforward because given an approximate counting oracle, one can determine the satisfiablity of F by the positivity of M. $M > 0$ is equivalent to F being satisfiable.

However, it is somewhat surprising that counting and approximate counting of some most restrictive classes of propositional logic are intractable in the worst case. One of such examples is 2MONCNF, a subset of both monotone CNF (all variables occur positively) and 2CNF. Determining satisfiability of 2MONCNF is trivial because each formula in 2MONCNF is satisfied by assigning truth value *True* to each variable. Counting the models of 2MONCNF is shown to be #P-complete, and it is NP-hard to approximate the count to a factor of $2^{n^{1-\epsilon}}$ for any positive constant ϵ [11]. Note that approximating to a factor of 2^n is equivalent to satisfiability problem.

Because of these strong negative results, for any interesting classes of propositional logic, guaranteed approximate counting algorithms that run in worst-case polynomial time are not expected to exist. There must be tradeoffs between the approximate factor guarantee and the worst-case run time guarantee. In practice, most existing algorithms sacrifice the run time guarantee to achieve a guaranteed approximation, which is often the exact count.

2.2 Counting by DPLL Searching

DPLL algorithm [4] was designed to find a satisfying assignment of a Boolean formula. The basic strategy of DPLL is that instead of looking for a satisfying assignment of the original formula F directly, one chooses any variable α that appears in F, and a satisfying assignment of either formula $(F \wedge \alpha)$ or formula $(F \wedge \neg\alpha)$ is also a satisfying assignment of F. The two sub-formulas can be simplified by unit-propagation, and solved recursively by calling DPLL algorithm.

The algorithm can be extended to a model counting algorithm in the following way [2]. Because the satisfying assignments of formula $(F \wedge \alpha)$ and formula $(F \wedge \neg\alpha)$ are disjoint,

$$\mathcal{M}(F) = \mathcal{M}(F \wedge \alpha) + \mathcal{M}(F \wedge \neg\alpha).$$

It is observed that when the two sub-formulas are simplified by unit-propagation, variables that appear in F may not appear in the simplified version of $(F \wedge \alpha)$ or $(F \wedge \neg\alpha)$. To expedite the search, if there are n variables in formula F, n^+ variables in Unitprop$(F \wedge \alpha)$, and n^- variables in Unitprop$(F \wedge \neg\alpha)$, then

$$\mathcal{M}(F) = \mathcal{M}(\text{Unitprop}(F \wedge \alpha)) \cdot 2^{n-n^+} + \mathcal{M}(\text{Unitprop}(F \wedge \neg\alpha)) \cdot 2^{n-n^-}.$$

In 2000, the idea of connected component analysis was introduced by Bayardo and Pehoushek [1] as an enhancement to Relsat's model counting ability. They draw connectivity graph for the formula. Nodes in the graph are variables in the formula. Two nodes are connected if the corresponding variables appear in the same clause. If a formula F can be decomposed to sub-formulas $F_1, F_2, \ldots, F_i$, which are induced by the connected components of its connectivity graph, then

$$\mathcal{M}(F) = \prod_{j=1}^{i} \mathcal{M}(F_j).$$

Therefore before picking a variable and branching on it as described previously, a formula is divided into components, and the models of each component is counted by a DPLL model counter recursively.

The most recent additions to DPLL model counting are the ideas of component caching and clause learning [12]. Component caching records the previous counted sub-problems, and therefore avoids counting the same components repetitively. Clause learning is much like that of ZChaff [9], where reasons of previously discovered conflicts are captured in new clauses, and the learned clauses are added to the original formula to avoid running into the same conflicts again. The introduction of these two powerful tools accelerate the model counting procedure by orders of magnitude.

2.3 Counting by Compiling

Another existing approach to model counting is to compile CNF formulas to logics for which counting operation is tractable. Examples of such logics include Ordered Binary Decision Diagrams (OBDD) [6], and its superset, Deterministic, Decomposable Negation Normal Form (d-DNNF) [3]. Model counting is a polynomial operation for both OBDD and d-DNNF, but CNFs cannot always be compiled to these forms of polynomial size. Although this approach is conceptually different from that of DPLL, the actual computation is often similar. For example, the search tree constructed by DPLL with component analysis can be view as a tree-structured d-DNNF, and the component caching idea corresponds to reuse of NNF fragments in non-tree-structured d-DNNF. Huang and Darwiche [6] show that DPLL algorithm can be used to compile CNFs to OBDDs efficiently, and techniques in SAT solvers, such as clause learning and unit propagation, can be utilized in the compiling process.

2.4 Our Proposal: Counting by Sampling

Since model counting problem is downward self-reducible [7], meaning that it can be solved using oracles to solve its sub-problems, approximate counting of solutions can be reduced to almost-uniform sampling of the solution space in polynomial time. ApproxCount algorithm is based on near-uniform sampling. One advantage of using sampling-based approximate counter is that the run time and accuracy are based on the number of samples the algorithm draws in each iteration. One can get a faster (and less accurate) approximation by reducing sample size. This is especially important in time-critical decision making. In searching and compiling based model counters, one cannot halt the execution and retrieve an approximation since it is very possible that many solutions reside in the final branches of the search tree, and, in case of connected component analysis, non-existence of solutions in the last component implies non-existent of solutions in the whole formula.

The best known methods for sampling from a predefined distribution in combinatorial space are Markov Chain Monte Carlo (MCMC) methods. MCMC methods set the target distribution as the stationary distribution of an ergodic Markov chain. With infinite time, the constructed Markov chain is guaranteed to reach its stationary distribution. However, in practice for complex combinatorial problems such as SAT, the Markov chain almost always takes exponential time to reach its stationary distribution [17]. The most widely used sampling algorithms such as Gibbs sampling are often trapped in modes (local minima) and does not converge in practical time limit.

For this reason, we built ApproxCount on SampleSat algorithm [17]. SampleSat is capable of sampling from solution space of a propositional logic near-uniformly and efficiently. We will first briefly review SampleSat algorithm, and then we will describe our implementation of an approximate counter based on it.

3 SampleSat

SampleSat algorithm is based on random walk strategies widely used in solving satisfiability problem [13, 18]. The inherent randomness in random walk style SAT solvers often leads the algorithms to different models in different runs. This provides a biased sampling of solution space. To reduce this bias, MCMC moves, more specifically, Metropolis moves are injected to interleave with random walk moves. This hybrid approach makes the sampling much more uniform, and is useful in many domains, including approximate model counting discussed in this paper.

3.1 Random Walk

Random walk (RW) as a local search heuristic was first proposed by Papadimitriou [10]. The algorithm starts from a random truth assignment. If the assignment has not already satisfied the formula, at each step, one unsatisfied clause is chosen uniformly at random. And then a variable in the clause is chosen by some heuristic **ChooseVar**. The value of the variable is flipped. The algorithm repeats these steps until a satisfying assignment is reached.

```
Procedure RW
repeat
        $c$:= an unsatisfied clause chosen at random
        $x$:= a variable in $c$ chosen by heuristic ChooseVar($c$)
        flip the value of $x$;
until a satisfying assignment is found.
```

Fig. 1. Random walk strategies

It has been shown that when **ChooseVar**(c) is "picking a variable in c uniformly at random", the RW procedure solves any 2CNF instance in quadratic time with high probability. For more general instances however, the algorithm needs to adopt a heuristic with greedy bias, which tries to satisfy more clauses on each flip. Therefore, the concept of "break value" of a variable is introduced. The break value of a variable is defined by the number of clauses that are currently satisfied but become unsatisfied when the truth value of the said variable is changed. In SampleSat, we follow the heuristic used in WalkSat [13], and define **ChooseVar** as in Figure 2.

```
Heuristic ChooseVar($c$)
if there exists a variable $x$ in $c$ with break value $= 0$
    return variable $x$
else
    with probability q
        $x$:= a variable in $c$ chosen at random;
        return variable $x$
    with probability (1-q)
        $x$:= a variable in $c$ with smallest break value
        return variable $x$
```

Fig. 2. Heuristic ChooseVar used in WalkSat and SampleSat

With multiple runs, we found that WalkSat is able to reach every solution of instances from many domains, such as random 3CNF, planning, and verification. However, the sampling is biased, especially for random 3CNF formulas. For example, we found in our experiments on a 70-variable random 3CNF formula near the transition point that the most frequently visited solution is reached 17,000 times more often than the least frequently visited solution. From the theoretically point of view, it is possible to construct a formula, for which a random walk strategy visits one model exponentially more often than it visits another model [17]. Intuitively, a model whose neighbors are all models of the formula cannot be reached by random walk unless it was hit by the initial guess of the algorithm. To reduce the bias and make the sampling more uniform, we incorporate Metropolis moves to the algorithm.

3.2 Metropolis

Metropolis moves are injected into random walk moves because of their favorable limiting properties. The limiting distribution of Metropolis algorithm is uniformly distributed over all models. Metropolis algorithm determines the change of a variable assignment by the decrease in the number of satisfied clauses, Δcost, caused by the change and a predetermined constant T representing temperature. The detail of our Metropolis move implementation is given in Figure 3.

```
Metropolis Algorithm
$x$:= a variable chosen uniformly at random;
if $\Delta\text{cost}(x) \leq 0$
    flip the value of $x$;
else
    with probability $e^{-\Delta\textbf{cost}/T}$
        flip the value of $x$.
```

Fig. 3. Metropolis moves

SampleSat combines random walk moves with Metropolis moves. At each step, the algorithm makes a random walk move with probability p, and it makes a Metropolis move with probability $(1 - p)$. Experiments show $p = 50\%$ yields good sampling results in many domains.

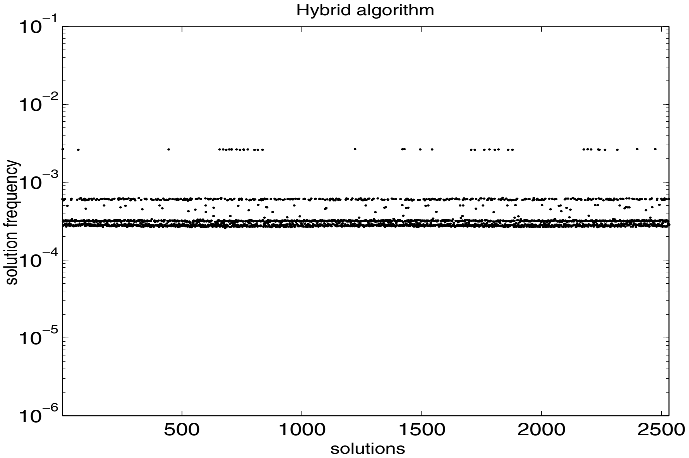


Fig. 4. Sampling of a hard 70-variable random 3SAT instance using SampleSat. Source: [17]

Figure 4 shows the sampling result of this hybrid algorithm on the 70-variable random formula mentioned in Section 3.1. The instance has 2531 solutions. Each dot in the figure presents one solution. The y-axis represents the solution frequency (#hits/#runs) of the solutions. We observe that the models are sampling quite uniformly. More specifically, the ratio between the highest solution frequency and the lowest solution frequency is reduced to a factor of around 10, compared to a factor of 17,000 when using the random walk strategy alone.

4 ApproxCount Algorithm

Since SampleSat produces efficient near-uniform sampling of the solution space, ApproxCount extends it to an approximate model counter based on the work of Jerrum *et al* [7]. The idea is to count the model of formula F, we first draw K samples from the solution space of F. (Note that a sample is a satisfying truth assignment.) The value of K is determined by the accuracy we want for our algorithm. If we consider a variable x_1 in F, and among the K samples, we denote the number of samples in which x_1 is assigned truth value *True* as $\#(x_1 = \text{True})$, and the number of samples in which x_1 is assigned truth value *False* as $\#(x_1 = \text{False})$. Assume the K samples are drawn from the uniform distribution over the models, and K is sufficiently large, then

$$\frac{\mathcal{M}(F \wedge x_1)}{\mathcal{M}(F)} \approx \frac{\#(x_1 = \text{True})}{K}, \quad \text{and} \quad \frac{\mathcal{M}(F \wedge \neg x_1)}{\mathcal{M}(F)} \approx \frac{\#(x_1 = \text{False})}{K}.$$

To ensure the stability of the algorithm, in each step we always pick truth value t such that $\#(x_1 = t) \geq \#(x_1 = \neg t)$. Without loss of generality, we assume t is *True*. The approximation above is equivalent to

```
ApproxCount Algorithm
repeat
    Draw $K$ samples from the solution space of $F$,
    $x$ := choose a variable in $F$ by PickVar($F$)
    Among these $K$ samples,
    if $\#(x = \text{True}) > \#(x = \text{False})$
        $F$ := Unitprop($F$, $x = \text{True}$)
        multiplier $M_x := K/\#(x = \text{True})$
    else
        $F$ := Unitprop($F$, $x = \text{False}$)
        multiplier $M_x := K/\#(x = \text{False})$
until $F$ = empty
output product of all multipliers.
```

Fig. 5. Approximate Counts

$$\mathcal{M}(F) \approx \frac{K}{\#(x_1 = \text{True})} \cdot \mathcal{M}(F \wedge x_1).$$

$M_{x_1} = \frac{\mathcal{M}(F)}{\mathcal{M}(F \wedge x_1)}$, called multiplier of variable x_1, can be approximated by $\frac{K}{\#(x_1 = \text{True})}$, and formula $(F \wedge x_1)$ is simplified with unit propagation and the simplified sub-formula $F_{x_1 = \text{True}}$ can be counted by the above procedure recursively. The overall calculation is

$$\begin{aligned}\mathcal{M}(F) &= \frac{\mathcal{M}(F)}{\mathcal{M}(F_{x_1=t_1})} \cdot \frac{\mathcal{M}(F_{x_1=t_1})}{\mathcal{M}(F_{x_1=t_1,x_2=t_2})} \cdot \;\cdots\; \cdot \frac{\mathcal{M}(F_{x_1=t_1,x_2=t_2,\ldots,x_{n-1}=t_{n-1}})}{1} \\ &= M_{x_1} \cdot M_{x_2} \cdot \;\cdots\; \cdot M_{x_n},\end{aligned}$$

where $t_1, t_2, \cdots, t_{n-1}$ are chosen such that the multipliers are always no greater than 2.

The outline of the algorithm is given in Figure 5. We have experimented with 4 heuristics for **PickVar** in the algorithm:

- *pickbiased*: always pick the variable that maximizes $|\#(x = \text{True}) - \#(x = \text{False})|$. The variables first selected by this heuristic are likely backbone variables, and setting them to their right values may help simplify the whole formula.
- *pickunbiased*: always pick the variable that minimizes $|\#(x = \text{True}) - \#(x = \text{False})|$. The variables first selected by this heuristic are likely don't-care variables, and setting them early may help the accuracy of counting.
- *pickrandom*: pick a variable uniformly at random.
- *pickbyorder*: first choose variable 1, and then variable 2, etc. In many structured problems, the variable order in the input formula often carries some domain information. For example, in graph coloring problems, the first variable may represent the first node is colored with the first color, etc. Following this order may help model counting. This heuristic is also designed for users who want to specify the order in which the variables should be set. To do

Table 1. We rank heuristics *pickbiased*, *pickunbiased*, and *pickrandom* according to their accuracy in a suite of 100 formulas. The suite includes 50 random formulas and 50 structured formulas

POSITION	*pickrandom*	*pickunbiased*	*pickbiased*
1ST PLACE	78	18	4
2ND PLACE	10	59	31
3RD PLACE	12	23	65

so, users just need to rename the variables in input formula according their desired order of assignment.

We did experiments to compare the effectiveness of the first three heuristics. (The last one is fully user-definable and encoding dependent, and will require separate study.) Table 1 shows that *pickrandom* clearly dominates the other 2 heuristics. In all experiments that we will show in the next section, we used heuristic *pickrandom*.

The run time of ApproxCount algorithm is upper-bounded by the product of the number of variables n, the number of solutions drawn in each iteration K, and the time needed to draw a solution c. Interesting and hard solution counting problems are usually under-constrained and have many solutions[1]. For these problems, c does not increase drastically with the problem size, and ApproxCount has a polynomial run time with regard to the problem size. In the left pane of Figure 7, we show the run time of ApproxCount for a class of synthetic formulas.

5 Experimental Results

We tested our algorithm on formulas from a variety of domains. It is well-known in satisfiability testing that algorithms' performance depends heavily on the problem structures. This is also true for model counting. Algorithm that performs well in one domain may not work well in other domains.

5.1 Random Formulas

There has been much interest in counting the satisfying assignments of random 3CNF formulas. Random formulas are generated for different clause/variable (C/V) values. Unlike in satisfiability testing, where different algorithms all experience the peak of difficulty at phase transition point around C/V = 4.26, the peak of difficulty for 3CNF model counting apparently shifts with the development of algorithms. Birnbaum and Lozinskii [2] report the peak of difficulty at C/V = 1.2 with their DPLL based CDP algorithm. Bayardo and Pehoushek [1] incorporate connected component analysis, and report the peak of difficulty at C/V = 1.5 with their DDP algorithm. Sang *et al* [12] add clauses learning and component caching to component analysis, and find the peak at the ratio of 1.8.

[1] We will discuss more about this in Section 5.1.

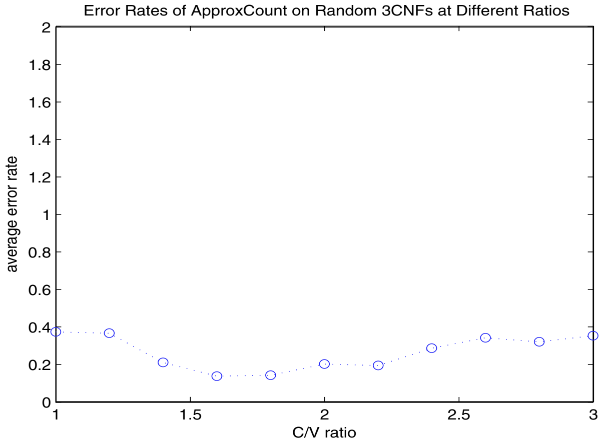


Fig. 6. Average error rates for ApproxCount on random 3CNF formulas with 75 variables. 1000 samples were drawn in each iteration. 20 instances were generated at each ratio. Exact model counts of the instances were calculated by Cachet [12]

Darwiche [3] compiles random 3CNF formulas to d-DNNFs, and also finds the peak at C/V = 1.8. Huang and Darwiche [6] use DPLL to compile these formulas to OBDDs, and find the peak at C/V = 2.0. The peak of difficulty is away from the peak of satisfiability test at 4.26 for these algorithms because instead of stopping at the first model, the algorithms need to visit every branch of the search tree with models. Formulas with less constraints have many more models, and make the counting harder.

For ApproxCount algorithm, counting models of formulas with the same number of variables and different ratios consumes almost the same amount of time[2]. Figure 6 shows the quality of approximation for different C/V ratios. The x-axis is the C/V ratio for random 3CNF formulas, and y-axis is the average error rate. If the approximate model count is a, and exact model count is t, the error rate is defined as $\frac{|a-t|}{\min(a,t)}$. 20 instances were generated at each ratio. From the figure, we see the error rates are quite low considering that approximation is NP-hard in the worst-case. We also observe that the error does seem relatively independent of the C/V ratio.

5.2 Application Domains

We also tested ApproxCount on structured formulas. Most of these formulas are taken from SATLIB [5]. The counting results are given in Table 2. In most of these domains, ApproxCount approximates the true model count within a factor

[2] Actually higher ratio formulas have more clauses, and make ApproxCount spend slightly more time than lower ratio formulas, but the difference is very small.

Table 2. ApproxCount results in application domains

<table>
<tr><th colspan="5">All Interval Series</th></tr>
<tr><td>instance</td><td>#var</td><td>#clauses</td><td>#models</td><td>ApproxCount result</td></tr>
<tr><td>ais6</td><td>61</td><td>581</td><td>24</td><td>24</td></tr>
<tr><td>ais8</td><td>113</td><td>1520</td><td>40</td><td>40</td></tr>
<tr><th colspan="5">Circuit Fault Analysis</th></tr>
<tr><td>instance</td><td>#var</td><td>#clauses</td><td>#models</td><td>ApproxCount result</td></tr>
<tr><td>ssa7552-158</td><td>1363</td><td>3064</td><td>25×10^{30}</td><td>7×10^{30}</td></tr>
<tr><td>ssa7552-159</td><td>1363</td><td>3032</td><td>7×10^{33}</td><td>3×10^{33}</td></tr>
<tr><th colspan="5">Graph Coloring</th></tr>
<tr><td>instance</td><td>#var</td><td>#clauses</td><td>#models</td><td>ApproxCount result</td></tr>
<tr><td>flat30-3</td><td>90</td><td>300</td><td>1968</td><td>2422</td></tr>
<tr><td>flat30-4</td><td>90</td><td>300</td><td>720</td><td>985</td></tr>
<tr><td>flat30-5</td><td>90</td><td>300</td><td>1362</td><td>1338</td></tr>
<tr><th colspan="5">Boolean Vector</th></tr>
<tr><td>instance</td><td>#var</td><td>#clauses</td><td>#models</td><td>ApproxCount result</td></tr>
<tr><td>2bitcomp_5</td><td>125</td><td>310</td><td>9.8×10^{15}</td><td>3.6×10^{15}</td></tr>
<tr><td>2bitcomp_6</td><td>252</td><td>766</td><td>21×10^{28}</td><td>3.8×10^{28}</td></tr>
<tr><th colspan="5">Logistics</th></tr>
<tr><td>instance</td><td>#var</td><td>#clauses</td><td>#models</td><td>ApproxCount result</td></tr>
<tr><td>prob004-log-a</td><td>1790</td><td>18026</td><td>2.6×10^{16}</td><td>1.4×10^{16}</td></tr>
<tr><th colspan="5">Bounded Model Checking</th></tr>
<tr><td>instance</td><td>#var</td><td>#clauses</td><td>#models</td><td>ApproxCount result</td></tr>
<tr><td>dp02s02.shuffled</td><td>319</td><td>683</td><td>1.5×10^{25}</td><td>1.2×10^{25}</td></tr>
</table>

of 2 or better. There are some domains, such as Boolean Vector, that are harder than others for ApproxCount.

5.3 Synthetic Domains

So far we have discussed instances that DPLL-based model counters can count exactly in order to evaluate the accuracy of our approximate model counter. However, ApproxCount can also provide good estimates for formulas that DPLL-based counters cannot count in reasonable time and memory limits. To show this, we need to design a class of formulas that we know the exact model counts by other means.

For this reason, we encode the following combinatorial problem to Boolean logic: suppose there are n different items, and you want to choose from the n items a list (order matters) of m different items ($m \leq n$). Let $P(n, m)$ represent the number of different lists you can construct. The value of $P(n, m)$ is given by $P(n, m) = \frac{n!}{(n-m)!}$.

The encoding to propositional logic works as follows. For each position i, and each item j, we use a Boolean variable $x_{i,j}$ to represent "whether position i will hold item j". We have the following three kinds of clauses:

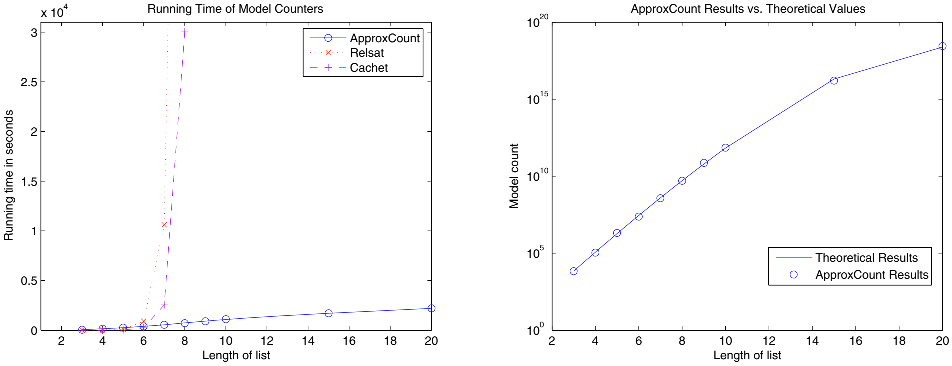


Fig. 7. Performance of model counters for combinatorial problem $P(20, m)$. In both figures, x-axes represent the value of m

- "each position holds at most one item": this translates into $mn(n-1)/2$ clauses
$$\neg x_{i,j_1} \vee \neg x_{i,j_2}, \quad \text{for any } i, j_1 > j_2;$$
- "each position holds at least one item": this translates into m clauses
$$\vee_{j=1}^{n} x_{i,j}, \quad \text{for any } i;$$
- "different positions hold different items": this translates into $nm(m-1)/2$ clauses
$$\neg x_{i_1,j} \vee \neg x_{i_2,j}, \quad \text{for any } j, i_1 > i_2.$$

Therefore, each instance of the problem is encoded to a Boolean formula with mn variables and $(mn(m+n-2)/2+m)$ clauses. These formulas seem hard for DPLL-based counters. As shown in the left pane of Figure 7, with the increase of length of the list, the run time of both Relsat [1] and Cachet [12] grows exponentially. The run time of ApproxCount is polynomially related to the problem size. In the right pane, we see that the estimates produced by ApproxCount are very close to the theoretical results.

We have experimented with larger formulas in the class than those given in Figure 7 to see how the algorithm scales. The results are given in Table 3. From the table, we see the algorithm scales well on this class of formulas. Because ApproxCount calculates the total count by the product of estimated multipliers of each variable, the error at each step could potentially accumulate to a large overall error. To understand why this does not happen, we calculate the error rate of ApproxCount at each step of a small formula $P(20, 4)$ using Cachet, and find the average error rate (as defined in Section 5.1) of each estimated multiplier is 0.71%, with a standard deviation of 0.48%. However, in about 50% of the steps ApproxCount over-estimates the multipliers, and in the other 50% of the steps it under-estimates the multipliers. Overall these errors cancel out each other for the most part, and result in an overall error rate of around 5%. If we distribute

Table 3. ApproxCount results on $P(n, m)$. The last column gives the average error in each step that actually contributes to the final error

n	m	#var	#clauses	#models	ApproxCount result	error per step
20	10	200	2810	6.7×10^{11}	7.2×10^{11}	0.05%
20	15	300	4965	2.0×10^{16}	1.6×10^{16}	0.09%
20	20	400	7620	2.4×10^{18}	2.8×10^{18}	0.04%
25	10	250	4135	1.2×10^{13}	1.1×10^{13}	0.04%
25	15	375	7140	4.3×10^{18}	4.8×10^{18}	0.03%
25	20	500	10770	1.3×10^{23}	1.4×10^{23}	0.02%
30	10	300	5710	10.9×10^{13}	9.1×10^{13}	0.07%
30	15	450	9690	2.0×10^{20}	2.1×10^{20}	0.01%
30	20	600	14420	7.3×10^{25}	5.9×10^{25}	0.04%

this error to the 80 steps, each step only contributes 0.06% to the final error, and other part is canceled by the opposite errors at other steps. For larger formulas, however, we can no longer calculate error rate at each step. In the last column in Table 3, we calculate the average error rate at each step that contributes to the final error. This error rate is consistently low across all sizes of formulas in the table.

6 Dealing with Real Numbers in Probabilistic Reasoning

In many probabilistic reasoning models, such as in Bayesian networks, probabilities are represented by real numbers, therefore the reasoning tasks can often be converted naturally to weighted model counting of Boolean formulas [12][3]. In weighted model counting, each variable a is assigned a weight $w_a \in [0, 1]$, and its negation $\neg a$ is assigned weight $1 - w_a$. $w(a = t)$ is defined as w_a when t is *True*, and $1 - w_a$ when t is *False*. The weight of a model is the product of the weights of its literals. The weighted model count of a formula F, noted as $\mathcal{M}_w(F)$, is the sum of the weights of all of its models.

Unweighted counting we discussed in previous sections can be considered as a special case of weighted counting where each variable has a weight 0.5. We can use pure propositional logic to encode these real number weights. For example, if a variable a has a weight 0.375 we can encode a as $(a_1 \wedge a_2) \vee (a_3 \wedge a_4 \wedge a_5)$, and use unweighted model counter to count the model. The weighted model count can be calculated as the model count of the encoded formula divided by 2^n, where n is the number of variables. However, this approach introduces many new variables, and makes the counting inefficient.

ApproxCount algorithm can be easily modified to calculate weighted model count. Because

[3] Discussion about the conversion is available at http://www.cs.washington.edu /homes/kautz/talks/counting-sat04.ppt.

```
Weighted ApproxCount Algorithm
product := 1;
repeat
    Draw $K$ samples from the solution space of $F$,
    $x$ := choose a variable in $F$ by PickVar($F$)
    Among these $K$ samples,
    if $\#_w(x = \text{True}) > \#_w(x = \text{False})$
        $F$ := UnitProp($F$, $x = \text{True}$)
        multiplier $M_x := w(K)/\#_w(x = \text{True})$
        product := product * multiplier * $w(x = \text{True})$
    else
        $F$ := UnitProp($F$, $x = \text{False}$)
        multiplier $M_x := w(K)/\#_w(x = \text{False})$
        product := product * multiplier * $w(x = \text{False})$
until $F$ = empty
output product.
```

Fig. 8. Approximate Weighted Model Counts

$$\begin{aligned}
&\mathcal{M}_w(F) \\
&= \Big(\frac{\mathcal{M}_w(F)}{\mathcal{M}_w(F_{x_1=t_1}) \cdot w(x_1 = t_1)} w(x_1 = t_1)\Big) \\
&\quad \cdot \Big(\frac{\mathcal{M}_w(F_{x_1=t_1})}{\mathcal{M}_w(F_{x_1=t_1, x_2=t_2}) \cdot w(x_2 = t_2)} \cdot w(x_2 = t_2)\Big) \cdot \\
&\quad \cdots \cdot \Big(\frac{\mathcal{M}_w(F_{x_1=t_1, x_2=t_2, \ldots, x_{n-1}=t_{n-1}})}{1 \cdot w(x_n = t_n)} \cdot w(x_n = t_n)\Big) \\
&= \big(M_{x_1} \cdot w(x_1 = t_1)\big) \cdot \big(M_{x_2} \cdot w(x_2 = t_2)\big) \cdot \ \cdots \ \cdot \big(M_{x_n} \cdot w(x_n = t_n)\big).
\end{aligned}$$

At each step, the multiplier is estimated by the sum of weights of all samples divided by the sum of weights of samples that assign truth value t to the variable in consideration, where t is chosen such that the multiplier is no greater than 2 to maintain the stability of the estimate.

Figure 8 gives the modified ApproxCount algorithm. In the algorithm, $w(K)$ represents the sum of weights of the K samples drawn, and $\#_w(x = \text{True/False})$ represents sum of weights of samples that assign x to *True*/*False* among the K samples drawn.

7 Conclusion

We have presented ApproxCount algorithm that approximates the model count of a formula in propositional logic. We have shown that ApproxCount generates good estimates for formulas in several domains. The approach extends the range of formulas whose models can be counted approximately. ApproxCount proceeds incrementally, setting one variable at a time and computing a multiplier at each step. Most interestingly, our work suggests that the individual errors in the multipliers have a tendency to cancel out, thereby making the approach a very promising one. We also proposed modifications of the algorithm to deal with real numbers in probabilistic reasoning models.

Acknowledgements. This work was supported in part by an Alfred P. Sloan Research Fellowship, a National Science Foundation Information Technology Research Grant IIS-0312910, and the Intelligent Information System Institute at Cornell University sponsored by Air Force Office of Scientific Research Grant F49620-01-1-0076.

References

1. R. J. Bayardo, Jr. and J. D. Pehoushek. Counting Models Using Connected Components. In *Proc. AAAI-00*, 2000
2. E. Birnbaum and E. L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *Journal of Artificial Intelligence Research*, 10:457-477, 1999.
3. A. Darwiche. A Compiler for Deterministic, Decomposable Negation Normal Form. In *Proc. AAAI-02*, 2002.
4. M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5:394-397, 1962.
5. H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In *I. P. Gent, H. v.Maaren, T. Walsh, editors, SAT 2000* , pp.283-292, IOS Press, 2000. SATLIB is available online at www.satlib.org.
6. J. Huang and A. Darwiche. Using DPLL for Efficient OBDD Construction. In *Proc. SAT-04*, 2004
7. M. R. Jerrum, L. G. Valiant, V. V. Vazirani. Random Generation of Combinatorial Structures from a Uniform Distribution. *Theoretical Computer Science* 43, 169-188, 1986.
8. H. Kautz and B. Selman. Planning as Satisfiability. In *Proceedings ECAI-92*, 1992.
9. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering a Highly Efficient SAT Solver. *38th Design Autom. Conference (DAC 01)*, 2001.
10. C. H. Papadimitriou. On Selecting a Satisfying Truth Assignment. In *Proceedings of the Conference on the Foundations of Computer Science*, pages 163-169, 1991.
11. D. Roth. On the Hardness of Approximate Reasoning. *Artificial Intelligence* 82, 273-302, 1996.
12. T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *Proc. SAT-04*, 2004.
13. B. Selman, H. Kautz, and B. Cohen. Local Search Strategies for Satisfiability Testing. *2nd DIMACS Challenge on Cliques, Coloring and Satisfiability*, 1994.
14. S. Toba. PP is as Hard as the Polynomial-Time Hierarchy. *SIAM Journal on Computing*, Vol. 20, No. 5, 865-877, 1991.
15. L. G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, Vol. 8, No. 3, 410-421, 1979.
16. M. N. Velev and R. E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. *Journal of Symbolic Computation*, Vol. 35, No. 2, pp. 73-106, 2003.
17. W. Wei, J. Erenrich, and B. Selman. Towards Efficient Sampling: Exploiting Random Walk Strategies. In *Proc. AAAI-04*, 2004.
18. W. Wei and B. Selman. Accelerating Random Walks. In *Proc. CP-02*, 2002.
19. SAT Competition. http://www.satcompetition.org

Benchmarking SAT Solvers for Bounded Model Checking

Emmanuel Zarpas

IBM Haifa Research Laboratory,
zarpas@il.ibm.com

Abstract. Modern SAT solvers are highly dependent on heuristics. Therefore, benchmarking is of prime importance in evaluating the performances of different solvers. However, relevant benchmarking is not necessarily straightforward. We present our experiments using the IBM CNF Benchmark on several SAT solvers. Using the results, we attempt to define guidelines for a relevant benchmarking methodology, using SAT solvers for real life BMC applications.

1 Introduction

Over the past decade, formal verification via model checking has evolved from a theoretical concept to a production-level technique. It is being actively used in chip design projects across the industry, where formal verification engineers can now tackle the verification of large industrial hardware designs. Bounded Model Checking (BMC) [1] has recently enabled the verification of problems that used to be beyond the reach of formal verification. However, BMC performance relies heavily on the performance of the underlying SAT solver.

We conducted several experiments using the IBM CNF benchmark [12, 11] to evaluate SAT tools. Often, we found discrepancies between our results and the experimental results found in the literature. For example, our results are not necessarily in line with the results of the SAT03 contest [4]. The main goal of this paper is to outline a methodology for benchmarking SAT solvers on Bounded Model Checking problems.

The paper is organized as follows: In Section 2, we present experimental results for some famous SAT solvers. Section 3 presents a new version of the IBM Benchmark and discusses correlation of the CNF characteristics and solvers performance. Section 4 contains a discussion on SAT benchmarking for BMC, where we try to learn from our experimental results in light of other experiments such as the SAT contests. Section 5 concludes the paper.

2 Experiments with IBM 2002 Benchmark: SAT Solver Comparison

Our experiments compared four famous SAT solvers: zChaff I (2001.2.17 version) [7] , zChaff II (zChaff 2004.5.13) [3], BerkMin561 [2] and Siege_v4 [8]. We used the

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 340–354, 2005.

2002 version of the IBM CNF Benchmark[12], whose CNFs are generated through BMC from the IBM Formal Verification Benchmark Library[11]. This library is a collection of models from real-life industrial hardware verification projects.

For each model, we used the CNF formulas (SAT_dat.k.cnf) for bound with k=1, 10, 15, 20, 25, 30, 35, 40, 45, 50. The time-out was set at 10,000 seconds on a workstation with 867841X Intel(R) Xeon(TM) CPU 2.40GHz, with 512 KB first level cache and 2.5 GB physical memory. We used the default configuration for each engine.

2.1 zChaff I vs. BerkMin561

We ran zChaff I (2001.2.17 version) and BerkMin561 on the 2002 version of the IBM CNF Benchmark[1]. Table 1 is a summary of the results obtained. More complete experimental results are presented in Table 8, where for each model in the table, we present the sum of the results of SAT_dat.k.cnf with k=1, 10, 15, 20, 25, 30, 35, 40, 45, 50 (*i.e.*, the BMC translation of each model for the different k). For more details, see the complete results in [13]. Because SAT solvers do not always behave in homogeneous manner (*Cf.* detailed results in [13] or table 2), the complete results analysis should not be disregarded.

Table 1. The results are displayed in seconds and include the timeout 10,000 seconds

	zChaff	BerkMin561
Total time (10000 sec time-out)	344765	414094
First (# of CNF where the engine is the fastest)	298	131
# of unsolved problems (timeout reached)	25	30
+ (# of CNF where the engine is the fastest by more than a minute and 20%)	67	32
First by model (# of model where the engine is the fastest)	28	18

The results show that zChaff and BerkMin561 achieved close results. On some CNFs, zChaff runs faster, and on others, BerkMin561 runs faster. In most cases, the differences in their performance is not very significant, the highest speed is not faster by more than one minute or 20%. However, while zChaff seems to perform slightly better overall, BerkMin561 gets better results than zChaff on the UNSAT CNFs. From these results, it is not possible to conclude that BerkMin561 performs better than zChaff. This is not consistent with what can be read in the literature or with the final results of the SAT03 contest (Cf. section 2.4).

This is not a very satisfying conclusion and the previous metrics do not tell us in a simple way how much faster zChaff is than Berkmin561[2]. We need some

[1] BerkMin561 was second in the SAT03 contest industrial benchmarks category in the SAT03 contest; the winner Forklift is not publicly available.

[2] It is very important to be able to give clear and concise values for results dissemination.

Table 2. The results are displayed in seconds for the CNFs from 18_rule model. The time-out was set to 10000 seconds. zChaff performs the worst for k=15, 20, 35; BerkMin561 performs the worst for k=30, 40, 50; Siege_v4 has the best performance, except for k=15, 25

18_rule CNF	Result	zChaff	BerkMin	Siege
SAT_dat.k1.cnf	unsat	0	0	0
SAT_dat.k10.cnf	unsat	1	1	0
SAT_dat.k15.cnf	unsat	13	4	5
SAT_dat.k20.cnf	unsat	65	47	41
SAT_dat.k25.cnf	unsat	951	109	251
SAT_dat.k30.cnf	sat	700	755	557
SAT_dat.k35.cnf	sat	time-out	2230	1730
SAT_dat.k40.cnf	sat	5510	8060	247
SAT_dat.k45.cnf	sat	time-out	time-out	959
SAT_dat.k50.cnf	sat	5010	time-out	1370

additional metrics in order to compare these SAT solvers. The arithmetic mean of the speedup seems irrelevant. Let's say we want to compare the performances of solvers A and B. If the speedup of A vs. B is 10 on test_1 and 0.0001 on test_2, the arithmetic mean of the speedup (A vs. B) would be 5 and the mean of the speedup (B vs. A) would be 5000. This is clearly not relevant. The global speedup (Total time A / Total Time B) is good to have but the "long runs" have far more weight than the short ones. The median is much more interesting. however it does not take into account the extreme values (*e.g*, if on test_1 the speedup is equal to $median + 10$ or to $median + 0.1$, it does not change the median value). On the other hand, the median insensivity to extreme values makes it less dependent on the time-outs. The geometrical mean seems to be a good solution. It definitely takes into account the weight of the "negative speedup" (*i.e.*, a speedup of less than 1)[3].

We felt these values should be computed these values only on a relevant subset of the benchmark. For example, we discarded all the cases where the four SAT solvers time-out. In addition, we discarded cases for which the four solvers (zChaffI, Berkmin561, Siege_v4, zChaffII) run in less than five seconds. The rational is that it is difficult to accurately compare very small runtimes and that these very small runtimes are arguably not relevant. The results displayed in Table 3 (b) show that Berkmin561 is nearly two times slower than zChaff I, but that Berkmin behaves somehow better in the "long" cases.

2.2 Siege_v4

Siege was hors-concours for the SAT03 contest, therefore it did not participate in the second stage. Nevertheless, the Siege results were pretty good for the

[3] Note that the logarithm of the geometrical mean is equal to the arithmetical mean of the logarithms.

Table 3. With a 10000 sec. time-out

	Total Time (in seconds)	#time-outs
zChaff I	344,764	25
berkmin561	414,035	30
Siege_v4	197 239	14
zChaff II	389,304	31

(a)

Speedup	$\frac{zChaffI}{Berkmin}$	$\frac{zChaffI}{Siege}$	$\frac{zChaffI}{zChaffII}$
median	0.55	2.90	1.23
geomean	0.41	2.87	1.29
global	0.75	3.58	0.75

(b)

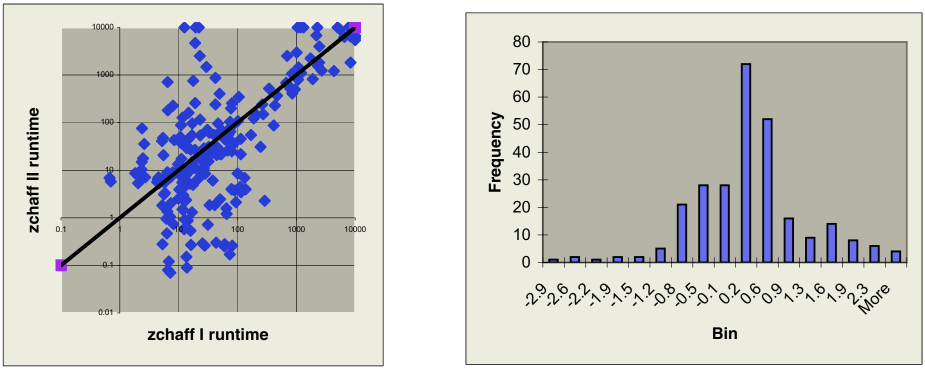
first stage and Siege has a reputation for being one of the best SAT solvers available. We ran Siege_v4[8] on the benchmark using 123456789 as a seed. Table 3 displays the overall conclusions. For more details see Table 8 and [13]. Siege_v4 is the fastest in 298 cases (CNFs) out of 498. In many difficult cases, Siege_v4 is fastest by an order of magnitude, or more. In some cases, Siege_v4 performed significantly worse than zChaff or BerkMin561. For example, for 26_rule, Siege_v4 is slower than zChaff by an order of magnitude and slower than BerkMin by two orders of magnitude. In addition, within the time-out, several CNFs can be solved only by Siege_v4 [4]. In conclusion, we see that Siege_v4 performs significantly better (roughly speaking 2.5 times faster) than zChaff I, Berkmin561 and zChaff II on the IBM CNF 2002 benchmark.

2.3 zChaff II

The industrial category of the SAT04 competition [6] was won by zChaff II (zChaff 2004.5.13). However "black-box" solvers such as Forklift (the 2003 edition winner) were "hors-concours" and as such not allowed to enter the second stage of the competition. We focus here on the performance of zChaff II vs zChaff I. zChaff II is faster than: zChaff for 229 CNFs (out of 442), Berkmin561 for 208 CNFs (out of 442), Siege_v4 for 82 CNFs (out of 442). In addition, zChaff II reached time-out more often than zChaff I (for six more cases). Table 3 displays a synthetic view of the results. The zChaff II code was left unchanged, so it used random seeds. Siege_v4 is roughly three time faster than zChaff I. Figure 1 gives us a graphic view of the performance of zChaff II vs. zChaff I. The overall performance difference between zChaff I and zChaff II on our benchmark is small[5], even though the two solvers can behave very differently in specific cases.

[4] Siege is a randomized solver, therefore, it could be argued that the comparison is not fair since the zChaff and BerkMin561, versions we used were not randomized. Nevertheless, even a deterministic solver can be lucky and providing Siege with a seed of Siege makes it deterministic.

[5] It should be note that zChaff II is randomized, so the results could be different for other runs. It would be interesting to have a statistical description of the range of zChaff II performances.



(a) (b)

Fig. 1. Histogram (b) gives the distribution of the decimal logarithm of zChaff I/zChaff II speedup with 10000 sec time-out

2.4 Comparison with SAT Contest Results

In order to understand whether or not our results are consistent with those of the SAT03 contest[6] (for details results see [5]), we had a look at the results of the first and second stages of the competition.

First stage. Looking at the results[7] for the industrial category [5], we note the following:

- Series *13_rule_*[8] is over-represented (Cf. Table 4). Besides, since all solvers time-out on most of the CNFs from this series, it is not very meaningful.
- Except for series *13_rule_* and *11_rule_*, the series from the IBM CNF benchmark are easy for zChaff, BerkMin561, and Siege_v1.
- Results for *rule_07* are not consistent with our own experiences. This discrepancy is probably caused by the "Lisa syndrome" [4]: CNFs were shuffled for SAT03 and solver performance can differ dramatically between a shuffled CNF and the original.

The SAT03 contest results for BerkMin561, Forklift, Siege_v1, and zChaff on (shuffled) series from the IBM CNF Benchmark are summarized in Table 4. If results from series *07_rule* and *13_rule_* are discarded, zChaff shows better results than BerkMin561.

[6] In the first stage of the SAT04 competition, zChaff I and zChaff II have very close results and the solvers that outperformed them (Forklift, Oepir) are not available for experimentation.

[7] The Siege version used for the SAT03 contest is Siege_v1.

[8] The names of the SAT03 series appears in *italics* in order to differentiate them from the series we used in our experiments. For the exact composition of the SAT03 series, see [5].

Table 4. Partial results from first stage SAT03 contest

	BerkMin561	forklift	Siege_1	zChaff
Total # of Solved Benchmarks	112	112	112	101
Total CPU time needed (sec)	105000	103000	103000	114000
without *13_rule_* (sec)	1510	518	408	4160
without *13_rule* and *07_rule* (sec)	1410	424	268	553

We believe that the differences in results for the first stage of SAT03 contest results for IBM the benchmark and our experiments are due to clause shuffling in SAT03 and to the fact that the SAT03 experiment used a smaller test-bed of CNFs from the IBM benchmark.

Second stage. During the second stage of the competition, solvers were ranked according to the number of CNFs they could solve from a restricted benchmark. Forklift was ranked first, Berkmin561 second, and zChaff I sixth on the industrial benchmark[9].

Clearly, the respective zChaff I and BerkMin561 ranking do not correspond to our experimental results (see Table 3). However, in the second stage, all solvers "timed-out" on the IBM benchmarks selected. In other words, the ranking of the solvers selected for the second stage of the competition did not take into account performance on the IBM benchmarks. This probably explains why our evaluation of zChaff and BerkMin561 on the IBM CNF Benchmark gives results that are not in line with the second stage results (on industrial benchmarks) of the SAT03 contest.

3 Experiments with IBM 2004 Benchmark: Search for Correlation

3.1 Description of the IBM 2004 Benchmark

Bounded Model Checking translation technology is continuously evolving. Therefore, we re-generated the IBM CNF Benchmark from the IBM Model Benchmark using an alternative BMC tool. We ran the new translation for the following bounds $k = 0..10, k = 11..15, k = 16..20, k = 21..25, \ldots, k = 95..100$. The 2004 CNFs are available on-line. In this paper, we refer to the 2002 and 2004 versions as the old and the new benchmarks, respectively .

Table 5 and 6 present statistical descriptions of the old and the new benchmarks. In these two tables, average is the arithmetical mean, STDEV is the standard deviation, $\frac{clauses}{var}$ is the ratio between the number of clauses and the number of variables, %unit is the percent of unit (*i.e.*, of length one) clauses in a CNF, %bin is the percent of clauses of length two, %ter is the percent of clauses

[9] Siege was hors-concours, therefore not selected for the second stage.

Table 5. Old (2002) benchmark

	var	clauses	$\frac{clauses}{var}$	%unit	%bin	%ter	%l=4	$\%l > 4$
average	80,167	343,826	4.12	0.3	75.5	14.2	3.5	6.5
median	54,857	220,180	4.08	0.2	77.0	12.2	3.5	6.6
STDEV	81,924	388,156	0.52	0.3	6.4	7.7	1.0	1.3
max	636,089	3,172,107	5.42	1.6	84.5	55.3	5.2	9.1
min	3,645	14,681	2.48	0	40.8	4.3	0.1	0.1

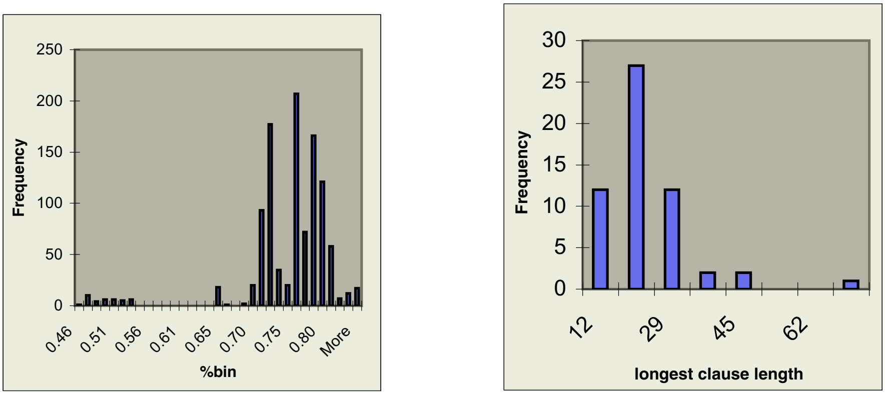
Table 6. New (2004) Benchmark

	var	clauses	$\frac{clauses}{var}$	%unit	%bin	%ter	%l=4	$\%l > 4$
average	73,414	305,301	3.99	0.6	74.5	15.0	3.6	6.3
median	50,897	195,612	3.98	0.4	76.1	13.1	3.6	6.6
STDEV	75,553	349,248	0.45	0.6	6.1	7.3	0.9	1.2
max	565,889	2,760,502	5.48	4.6	84.5	51.6	4.9	9.1
min	3,606	14,104	2.55	0	46.5	4.4	0.1	0.1

of length three, %l=4 is the percent of clauses of length 4 and $\%l > 4$ is the percent of clauses longer than 4.

These two tables tell us that the new benchmark CNFs are roughly 10% smaller than the old benchmark CNFs. However, they encompass roughly the same proportion of length two, three, and four clauses. The higher proportion of unit clauses in the new benchmark is due to specific optimizations. We also see that the ratio $\frac{clauses}{var}$ is slightly lower in the new benchmark (see the next subsection for a discussion on the relevance of this ratio). Figure 2 displays the histogram of %bin, where two models (07 and 13_1) have a "non standard" percent of binary clauses and are in the 45%-55% and not in the 70%-85%.

We looked at some statistics for the structure of the CNFs of the new benchmark. We found that for any model of the benchmark, there are four real numbers a, b, c, d such that for all the CNFs of the model, $\#(variables) \approx ak + b$ and $\#(clauses) \approx ck + d$, with k being the bound used to generate the CNFs. In a more general way, for any model and for any integer n strictly greater than 0, there are two real numbers e and f such that for all CNFs of the model #(clauses of length n) $\approx ek + f$ (of course e and f can be null, for example in most models, the number of unit clauses is a constant). The correlations (for a discussion about statistic techniques for NP-complete problems see [10]) between the series from the benchmark and the series predicted with the previous equations are about 0.99. This is not surprising since the CNFs are generated from the model in a way essentially linear to the bound. In addition, we found out that, in our benchmark, the length of the longest clause is the same for all CNFs generated from the same model. Figure 2 (b) displays the histogram of the longest clauses in the new benchmark. Note that two models, 07 and 13_1, which have the greater longest clauses, are also the ones with a "non standard" percent of binary clauses.



(a) (b)

Fig. 2. Histograms (a) and (b) give respectively the distribution of %bin and length of the longest clause

3.2 Experimentations and Correlations

How do zChaff I and Siege behave on the new benchmark? We ran zChaff I and Siege_v4 on both benchmarks (for $k = 10, 15, 20, \ldots, 50$) on the same workstation described in Section 1. Figure 3 displays the overall results.

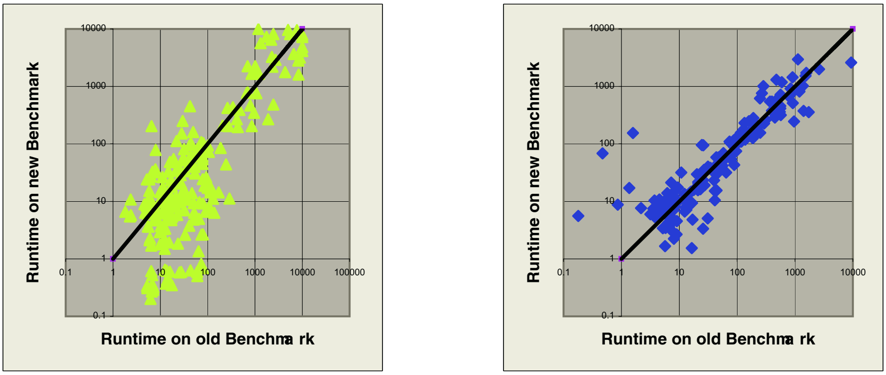
Strangely enough, the "long" cases of the new benchmark are more difficult for zChaff I but easier for Siege_v4. For both solvers, the correlation between performance on old and new benchmark is somewhat higher than 0.7. The correlation between the zChaff I and Siege runtimes on the new benchmark is also 0.72.

Can the "hardness" of a CNF be predicted from the value of the ratio (number of clauses)/(number of variables) ? We could not find any relevant correlation between this ratio and the zChaff I and Siege performances. In fact, as we saw previously, for each model, #clauses and #var are strongly correlated with the bound k; therefore, for each model, there are a, b, c, d four real numbers, such that, the ratio is also strongly correlated with $(ak + b)/(ck + d)$. This explains the appearance of Figure 4.

Can we predict the "hardness" of a CNF from the number of variables? This could also be the number of clauses, since both numbers are strongly correlated. In our experiments we found a relatively low (about 0.3) correlation between the number of variables and the runtimes for zChaff I and Siege. However, when we "standardized"[10] the runtimes, we got a higher correlation (about 0.7). See Figure 5 for illustration.

[10] For a given SAT solver, we standardized the runtimes in the following way: For each model, we divided each of the nine runtimes (corresponding to the nine bounds $k = 0..10, k = 11..15, k = 16..20, \ldots, k = 46..50$) by the greatest of these nine runtimes. The idea is to be able to compare results between different models.

Speedup	zChaff I	Siege
median	1.69	1.07
geomean	2.13	0.97
global	1.02	1.19



(a) (b)

Fig. 3. Performance comparison between the old and the new benchmark. Time-out: 10000 sec. (a) and (b) display zChaff I and Siege runtimes on the old benchmark vs. on the new benchmark. In both comparaisons, we discarded the cases which ran in less than 5 seconds in both cases or which time-out in both cases

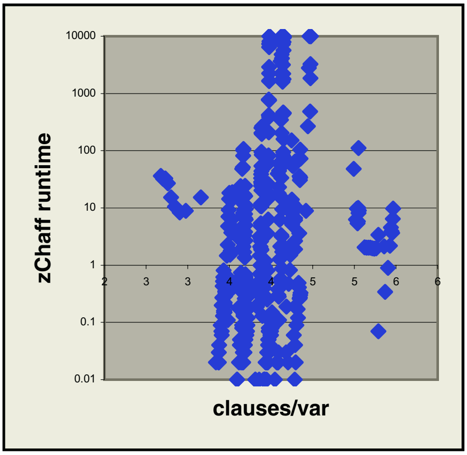
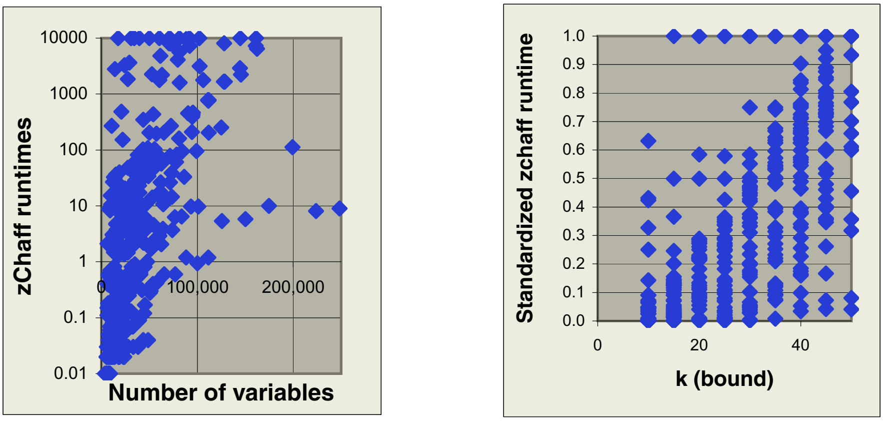


Fig. 4. $\frac{\#clause}{\#var}$ vs zChaff I runtime

Can we predict the "hardness" of a CNF from the proportion of clauses of length one, two, three and four ? We did not find any relevant correlation between the proportion of clauses of lenght two, three, four and more and the zChaff I and



(a) (b)

Fig. 5. (a): zChaff I runtime vs bound k, (b): standardized zChaff I runtime vs bound k

Table 7. Performance on SAT/UNSAT CNFs

		zChaff I	Siege	speedup
All CNFs	median	30	22	2.28
	geomean	56.52	31.80	1.75
	Total	192,323	44,632	
	global			4.31
UNSAT (109 CNFs)	median	18	12	2.46
	geoman	39	20	1.94
	total	77,120	14,642	
	global			5.27
SAT (67 CNFs)	median	55	59	1.84
	geoman	98	66	1.48
	total	115,203	29,989	
	global			3.84

Siege runtimes. We did find a low negative correlation (about -0.4) between the standardized proportion of unit clauses and the runtimes. We think this can be explained by the fact that for most models, the unit clause number is constant, so $\%unit \approx \frac{c}{ak+b}$, which explains the correlation.

Are "SAT" and "UNSAT" CNFs as hard? This is difficult to say since it is obviously not possible to compare performance on the same CNFs. It is not relevant to compare the geometrical mean of the runtime on the SAT and on the UNSAT CNFs, as displayed in Table 7. However, it is easier to compare the behavior of two SAT solvers on SAT and UNSAT CNFs. When we look at Table 7, we see that Siege behaves slower than to zChaff I on SAT CNFs as opposed to UNSAT CNFs.

4 Lessons for Benchmarking

Our experimental results, especially the comparison between zChaff and BerkMin561, are not in line with other results, such as those from SAT03[4]. In this section, we attempt to learn lessons from the previous experiments.

Use relevant benchmarks. To compare two SAT solvers for BMC, it is extremely important to use relevant benchmarks from BMC applications. For instance, results from theoretical problems such as 3-SAT problems, hand-made problems, or problems from applications others than BMC (such as planning), are not relevant and will not provide a good comparison between SAT solver performance for BMC applications. It is not sufficient to use only the CNFs generated by BMC; the models from which the CNFs are generated should be relevant. Therefore, it is of prime importance to use models from real-life industrial verification projects.

Use relevant time-out. The time-out used for the SAT03 contest was 600 seconds or 900 seconds for the first stage, and up to 2400 seconds for the second stage of the competition. For our experimentation, we used a 10000 second time-out. From the detailed experimental results [13], it is clear that the lower the time-out, the less relevant the results. Additionally, the lower the time-out, the more time-out results would be received for all the SAT solvers. Consequently, these time-outs can actually cover up different situations. For example, in our first experiments for CNF k45 from model 11_rule_1, with a time-out of 900 seconds, zChaff and BerkMin561 both timed out. However, with a 10000 second time-out, we realized that one solver performs seven times faster than the other on this CNF.

The ideal solution would be not to use any time-out or at least use very long time-outs (e. g., one week). In this way, the results for the SAT solvers can always (or almost always) be compared for a given CNF (at least a "reasonable" CNF). The obvious drawback is that this makes the experiments very, very long, hence limiting the scope of such experiments. In the SAT contests, the choice of a short time-out allows a great number of SAT solvers to be run against a very broad spectrum of CNFs. In our experiments, we ran a very limited number of SAT solvers against a smaller (but more relevant for BMC) number of CNFs with a longer time-out. We believe that this is one of the main explanations for the difference in results between SAT03 and our experiments. More precisely, if a longer time-out would have been used for the second stage of the SAT03 contest, some solvers would probably have solved the benchmark from the IBM CNF Benchmark; therefore, the final results would more likely have been consistent with our experiments. In summary, using shorter time-outs can be very useful for larger scopes experiments with many SAT solvers and very broad benchmarks. However, to get more relevant results for BMC[11], these kinds of experiments must be refined by limiting the scope (e. g., the number of SAT solvers).

[11] The SAT contest goals are not identical to ours. We are mainly interested in performance for real-life BMC problems, while the SAT contests try to establish "global" performance.

Shuffling clauses in benchmark CNFs is tricky. Shuffling clauses in benchmark CNFs can have a dramatic effect [4] and potentially change the ranking for solvers performance. However, in real life, SAT users don't shuffle their CNFs. One of the major differences between our experiments and SAT03 is that we did not shuffle CNFs.

Use a broad benchmark: easy for a SAT solver, does not mean easy for all. When assessing the performance of a new SAT solver, a common trend is to run the new solver against CNFs that are difficult to solve with a well-established solver (e. g., zChaff). Although it seems reasonable, this can be very misleading. Modern SAT solvers rely heavily on heuristics; therefore, a CNF can be solved very easily with one solver and still be very difficult with others. For example, the CNFs from model 01_rule are very easy for zChaff to solve but difficult for BerkMin561. On the 01_rule series, zChaff typically runs faster than BerkMin561 by two orders of magnitude. On the 26_rule series Siege_v4 typically runs slower than zChaff by one order of magnitude and slower than BerkMin561 by two orders of magnitude. The reason for this kind of behavior is not that some models are "special", but more likely the limitation of the heuristics used by different solvers. The fact that solver performance ranking often varies for different CNFs generated from the same model[12] comforts us with the idea that the relationship between solvers (and heuristics) performance and model specifics is not as strong as one would think.

Discard cases with unrelevant runtimes. It does not make a lot of sense to use a CNF C to, say, discriminate between two solvers A and B, if both A and B solve C in a very short period of time. Indeed, it is likely that the time sampling would not be accurate and relevant, especially because most solvers run on Unix/Linux machines. In the second part of our experiments, we took 5 seconds as a threshold.

For results analysis, there is no real reason to discriminate between results for satisfiable and unsatisfiable CNFs. We believe that it does not make much sense to differentiate between the performance of solvers for satisfiable and unsatisfiable CNFs – at least for BMC applications. Firstly, BMC is usually run in an incremental manner (e.g., $k = 0 \ldots 10, k = 11 \ldots 15, \ldots$). Therefore, before you can get a satisfiable result you often have to get several unsatisfiable results. Secondly, some models cannot be falsified and CNFs generated by BMC will always be unsatisfiable. Therefore, only the global performance is really relevant for real-life BMC SAT use.

Use several metrics for comparisons. Since SAT solvers rely heavily on heuristics, it is unlikely (or at least rare) that a SAT solver would be better on all possible CNFs (from real-life models). For example, even though Siege_v4 performs better than zChaff and BerkMin561 on most CNFs of the IBM benchmark, it does not perform better for all (e.g., model 26 in Table 8). The following metrics can be used to compare two SAT solvers, solver 1 and solver 2:

[12] In addition, Table 2 shows that the evolution between the bound k and the runtime is not the same for the three solvers.

Table 8. The results are displayed in seconds. The time-out was set to 10000 seconds. For each model, the number of time-outs, if any, appears in brackets

Model	zChaff	BerkMin561	Siege_v4	Model	zChaff	BerkMin561	Siege_v4
01_	166	22800 (1)	262	14__2	3490	2480	378
02_1__1	347	34	38	15_	6	21100 (1)	1
02_1__2	336	69	104	16_2__1	0	10	0
02_1__3	26	16	10	16_2__2	0	9	1
02_1__4	34	12	13	16_2__3	0	11	0
02_1__5	32	16	19	16_2__4	3	16	0
02_2_	118	83	11	16_2__5	1	17	2
02_3__1	96	157	22	16_2__6	1	18	1
02_3__2	702	34	13	17_1__1	0	509	0
02_3__3	154	133	18	17_1__2	219	427	70
02_3__4	333	39	13	17_2__1	1	28	0
02_3__5	126	172	25	17_2__2	1	98	0
02_3__6	396	34	16	18_	32200(2)	31200 (2)	5160
02_3__7	262	86	39	19_	127	2910	482
03_	190	733	126	20_	21800 (1)	9590	4020
04_	170	597	161	21_	150	2245	383
05_	33	267	124	22_	5290	5980	582
06_	296	1140	130	23_	38700 (2)	28500 (1)	2820
07_	61	73	69	26_	155	26	2850
09_	7	1	2	27_	53	364	134
11__1	5080	21900 (1)	1760	28_	385	843	96
11__2	6640	14500 (1)	982	29_	35700 (2)	58200 (5)	16700
11__3	24100 (2)	24000 (2)	3135	30_	76600 (7)	72000 (7)	66000(5)
12_	0	0	0	14__1	191	719	126
13__1	90000 (9)	90000 (9)	90000 (9)				

- Global times or the global speedup. This presents two drawbacks: 1) There is no perfect solution to take time-outs into account, and 2) All the weight is on the CNFs that take the longest to be solved.
- Ratio between the number of CNFs solved more quickly by solver 1 and the number of CNFs solved more quickly by solver 2) Optionally, only the cases where performance differences are significant can be taken into account. The median of the speedups is a better and more concise metric.
- The geometrical mean of the speedup is also a very interresting metric. However, it is also quite sensitive to the time-out definition.
- Time-out numbers. There are quite relevant if one considers that the time-out value is roughly equivalent to the real-life time-out (*i. e.*, the maximum reasonable waiting time of the real-life formal tool users). On other hand, this metric can easily give a biased view of reality. Imagine, for example, that the geometrical mean of the speedups for solver A vs. solver B. is 3, but solver A time-out slightly more often than solver B, should we really conclude that solver A is the slowest.

Each of these metrics have their pros and cons. We believe that the best solution is to use more than one of them to compare two SAT solvers. We believe that it is difficult to decide which solver is the best when, for example the geometrical mean, the median, and the global speedups give contradictory results.

5 Conclusion

In this paper, we showed that benchmarking is not a trivial task and that it can be misleading. For example, our experiments on zChaff and BerkMin561 present results that are contradictory with what is commonly accepted by the SAT community (*i.e.* BerkMin561 outperforms zChaff I). We also showed that it can be difficult to compare two SAT solvers (e. g., CNFs that are difficult for zChaff I are not the same as CNFs that are difficult for BerkMin561). Even when a SAT solver such as Siege_v4 seems to clearly outperform BerkMin561 and zChaff, it is not necessarily so for all benchmark CNFs. In order to help benchmarking and compare between solvers results, we proposed guidelines that sketch out a rough methodology. We believe that the systematic use of such a benchmarking methodology can improve the general quality of experimental performance evaluations for SAT and will help to produce practical and fundamental advances in the area. We plan to keep investigating possible correlations between CNFs static characteristics and the performances of SAT solvers.

Acknowledgment. The author wishes to thank Daniel Le Berre for his valuable comments and remarks on an earlier version of this paper.

References

1. Biere A.*et al*, “Symbolic Model Checking Without BDDs”. *Proc. of the workshop on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1579, 1999.
2. Goldberg E., Novikov Y., “BerkMin: a Fast and Robust SAT-solver” *Proc. of the Design, Automation and Test in Europe*, IEEE Computer Society, 2002.
3. Mahajan Y., Fu Z., Malik S., “Zchaff2004: An Efficient SAT Solver”. To appear in *SAT 2004* Special Volume.
4. Le Berre D., Simon L., ”The essentials of the SAT 2003 competition”, *Proc. of the 6th Int’l Conf. on Theory and Applications of satisfiability Testing*, LNCS 2919, 2003.
5. Le Berre D., Simon L., SAT2003 contest results. http://www.lri.fr/simon/contest03/results/
6. Le Berre D., Simon L., “55 Solvers in Vancouver: The SAT 2004 competition”. To appear in *SAT 2004* Special Volume.
7. Moskewicz M. *et al*, “Chaff: Engineering an Efficient SAT Solver”. *38th Design Automation Conference.* ACM/IEEE, 2001.
8. Ryan L., The siege satisfiability solver. http://www.cs.sfu.ca/ loryan/personal/

9. Shacham O., Zarpas E., “Tuning the VSIDS Decision Heuristic for Bounded Model Checking”. *Proc. of the 4th International Workshop on Microprocessor, Test and Verification*, IEEE Computer Society, 2003.
10. Van Gelder A.,Tsuji Y. K., ”Satisfiability Testing with More Reasoning and Less Guessing”. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.* American Mathematical Society, 1996.
11. Zarpas, E, “Simple yet efficient improvements of SAT based Bounded Model Checking”. *Proc. of Formal Methods in CAD: 5th Int'l Conf.*, LNCS, 3312, 2004
12. CNF Benchmarks from IBM Formal Verification Benchmarks Library. www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html
13. IBM CNF Benchmark Illustration. www.haifa.il.ibm.com/projects/verification/RB_Homepage/papers/comparaison2.zip

Model-Equivalent Reductions

Xishun Zhao[1],* and Hans Kleine Büning[2]

[1] Institute of Logic and Cognition,
Sun Yat-sen University,
510275 Guangzhou, P.R. China
hsdp08@zsu.edu.cn

[2] Department of Computer Science,
Universität Paderborn,
33095 Paderborn, Germany
kbcsl@upb.de

Abstract. In this paper, the notions of polynomial–time model equivalent reduction and polynomial–space model equivalent reduction are introduced in order to investigate in a subtle way the expressive power of different theories. We compare according to these notions some classes of propositional formulas and quantified Boolean formulas. Our results show that classes of theories with the same complexity might have different representation strength under some conjectures which are widely believed to be true in computation complexity theory.

1 Introduction

Generally speaking, a finite theory is a propositional formula, or a quantified Boolean formula, or a (disjunctive) logic program, and so on. Different theories may have different semantics. A model for a propositional formula is a subset of propositional variables such that its characteristic function is a satisfying truth assignment for the formula. A model for a quantified Boolean formula with free variables can be considered as a subset of free variables of the formula such that after replacing each free variable by its truth value the resulting formula is true. The well-known semantics for (disjunctive) logic programs is the stable model semantics. A theory is consistent if it has a model.

In representation theory, there are several notions for comparing the expressive power of different theories. One is the inference equivalence between theories, that is, to determine whether a theory can be represented as another theory such that they have the same inference ability. For inference equivalence, some theories with large size may be equivalent to theories with very small size and vice versa. For example, any propositional tautology is equivalent to $x \vee \neg x$; and some propositional formulas can be represented by quantified Boolean formulas with essentially small size. Another topic in representation theory is to

* This research was partially supported by the NSFC project under grant number:10410638

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 355–370, 2005.

determine whether theories can be transformed into other theories preserving the models. This approach may also have drawbacks. One is that the transformation may cost super-polynomial time; another is that the target theories may have super-polynomial size. It is well known that every propositional formula can be transformed into a formula in conjunctive normal form (CNF) such that they have the same models. However, there is a propositional formula φ such that any formula in CNF equivalent to φ has super-polynomial size. A theorem by Lin and Zhao [20] shows how to turn a normal logic program P to a propositional formula φ such that any model of φ is a stable model of P and vice-versa. The formula generated by Lin and Zhao's process can also be significantly larger than the original program. More recently, Lifschitz and Razborov have shown under the assumption $\mathrm{P}\not\subseteq\mathrm{NC}^1/\mathrm{poly}$ that any equivalent translation from logic programs to propositional formulas involves a significant increase in size.

That means, if we demand that transformations cost polynomial–time and preserve models, then classes of theories may have different expressiveness even if they have the same complexity. We shall introduce a reduction relationship to compare the representability of classes of theories. Let $\mathcal{C}, \mathcal{C}'$ be two classes of theories. We say $\mathcal{C}$ can be *polynomial–time model–equivalently reduced to* $\mathcal{C}'$ if every theory $T \in \mathcal{C}$ can be transformed in polynomial time into a theory $T' \in \mathcal{C}'$ such that there is a polynomial-time computable one-to-one correspondence between the models of T and T'.

If the consistency checking problem for $\mathcal{C}$ is harder than that for $\mathcal{C}'$, then it would be hopeless to construct a polynomial–time model–equivalent reduction from $\mathcal{C}$ to $\mathcal{C}'$. If we just require that the transformation from T to T' cost polynomial size instead of requiring that the transformation cost polynomial time then we say that $\mathcal{C}$ can be *polynomial–space model-equivalently reduced to* $\mathcal{C}'$.

The main results we obtain belong to the following categories.

1. For some classes $\mathcal{C}_1$ and $\mathcal{C}_2$ with the same complexity of the satisfiability problem (up to some polynomial), $\mathcal{C}_1$ can not even be polynomial–space model-equivalently reduced to $\mathcal{C}_2$. Let $\exists\mathrm{CNF}^*$ be the class of quantified Boolean formulas with only existential quantifiers and CNF kernels (here free variables are allowed). Clearly, the satisfiability problem for $\exists\mathrm{CNF}^*$ is also NP-complete, while $\exists\mathrm{CNF}^*$ can not be polynomial–space reduced to PF, the class of all propositional formulas. Such results are proved under some widely accepted conjectures on complexity classes.
2. Some classes $\mathcal{C}_1$ and $\mathcal{C}_2$ with the same complexity of the satisfiability problem (up to some polynomial), can be polynomial–time model–equivalently reduced to each other. For example, CNF can be polynomial–time model–equivalently reduced to 3CNF. That is, every propositional formula φ in conjunctive normal form can be model-equivalently transformed in polynomial time into a formula φ' in 3CNF. This is interesting because no polynomial–time transformation from CNF to 3CNF exists if we demand that φ and φ' be equivalent.
3. For some classes $\mathcal{C}_1$ and $\mathcal{C}_2$ for which the satisfiable problem for $\mathcal{C}_1$ has lower complexity than that for $\mathcal{C}_2$, $\mathcal{C}_1$ can not even be polynomial–space

model-equivalently reduced to $\mathcal{C}_2$. For example, the satisfiable problem for $\forall\neg$HORN* can be solved in polynomial time [13], while it can not be polynomial–space model–equivalently reduced to PF. Here $\forall\neg$HORN* is the class of quantified Boolean formulas with only universal quantifiers and kernels given as a negation of HORN formulas. These results are also subject to some conjectures on complexity classes.

Besides the insight into the power of the representation of various classes of formulas, our work may also find applications in practice. In knowledge representation, users may prefer to represent their knowledge base with required abilities in a concise way. QBF are also used in VLSI design specification. Since a piece of chip is extremely expensive the designer would like to design a circuit which can provide required functionality using a number of pieces as small as possible. Results of polynomial–time model–equivalent reduction may also allow us to use algorithms in one area to solve problems in other areas. More precisely, suppose $\mathcal{C}_1$ can be polynomial–time reduced to $\mathcal{C}_2$ and suppose we have had some efficient algorithms solving $\mathcal{C}_2$, then we could transform theories in $\mathcal{C}_1$ to those in $\mathcal{C}_2$ and then apply the known algorithms. The polynomial–space model–equivalent reduction might solve hard problems by doing off-line preprocessing. Suppose $\mathcal{C}_1$ can be polynomial–space model–equivalently reduced to $\mathcal{C}_2$ and a problem for $\mathcal{C}_1$ is hard while it is solvable in polynomial–time for $\mathcal{C}_2$, then we could transform off-line theories in $\mathcal{C}_1$ to those in $\mathcal{C}_2$, and then use algorithms for $\mathcal{C}_2$ to solve the problem on-line (for the notions of *off-line*, *on-line*, and *compilability*, please see [2]). Therefore, our research might be helpful and useful in knowledge representation and reasoning, circuit designing, model checking, and etc.

The remainder of this paper is organized as follows. In section 2, we recall some notions and terminologies which will be used later on. In sections 3, precise definitions of polynomial–time model–equivalent reduction and polynomial–space model-equivalence reduction are introduced. Section 4-5 are devoted to the investigation of model-equivalent relations between some classes of propositional formulas and some classes of quantified Boolean formulas with free variables. In section 6, we propose some open questions and some future work.

2 Preliminaries

2.1 Propositional Formulas and Quantified Boolean Formulas

In this paper, we use x, y, z, w to denote propositional (or Boolean) variables and L_i for literals (variables or their negation). Clauses are finite disjunctions of literals. PF is the class of propositional formulas and CNF (resp. DNF) is the class of propositional formulas in conjunctive (resp. disjunctive) normal form. The classes of propositional formulas such as kCNF, HORN and so on, are defined as usual. Given a class $\mathcal{C}$ of propositional formulas, $\neg\mathcal{C}$ is the class of formulas $\neg\varphi$ with $\varphi \in \mathcal{C}$. A *model* for a propositional formula φ is a subset M of variables occurring in φ such that the characteristic function is a satisfying truth assignment for φ.

QBF is the class of closed quantified Boolean formulas (every variable is quantified) in prenex normal form. That means, a quantified Boolean formula is of the form $Q_1x_1 \cdots Q_nx_n\varphi$, where $Q_i \in \{\forall, \exists\}$ and φ is a propositional formula over variables $x_1, \cdots, x_n$. $Q_1x_1 \cdots Q_nx_n$ is called the prefix and φ the kernel of the formula. Usually, we simply write $\Phi = Q\varphi$. A literal x or $\neg x$ is called a universal resp. existential literal, if the variable x is bounded by a universal quantifier resp. by an existential quantifier. A closed formula $\Phi \in$ QBF is true, if there exists an assignment of truth values to the existential variables depending on the truth values for the preceding universal variables, for which the propositional kernel of the formula is true. Given a class $\mathcal{C}$ of propositional formulas, $Q\mathcal{C}$ denotes the class of QBF formulas with kernel in $\mathcal{C}$, for example QCNF, QkCNF, and QHORN.

For a subclass $\mathcal{B}$ of QBF, the class $\mathcal{B}^*$ is defined in the same way as $\mathcal{B}$ except allowing free variables (i.e., variables in the kernel being not quantified). For example, QCNF* is the class of quantified Boolean formulas with CNF kernel and free variables. Propositional formulas can be considered as QBF* formulas with empty prefix.

For a class $\mathcal{B}^*$ of QBF* formulas and a natural number k, Σ_k-$\mathcal{B}^*$ (Π_k-$\mathcal{B}^*$, respectively) is the class of formulas in $\mathcal{B}^*$ with prefix type Σ_k (Π_k, respectively). Here, the prefix type $\Sigma_0 = \Pi_0$ is the empty prefix (no quantifiers). If a prefix Q has type Σ_k (Π_k, respectively), then the prefix $\forall \boldsymbol{x}Q$ ($\exists \boldsymbol{x}Q$, respectively) has type Π_{k+1} (Σ_{k+1}, respectively), where $\boldsymbol{x} = x_1, \cdots, x_n$, and $\forall \boldsymbol{x}$ is the abbreviation for $\forall x_1 \cdots \forall x_n$, likewise for $\exists \boldsymbol{x}$.

We often write a formula in QBF* as $\Phi(\boldsymbol{z})$ to state that free variables in Φ are among $\boldsymbol{z}$. $\Phi(\boldsymbol{z})$ is satisfiable if and only if there is a truth assignment v for the free variables, such that the closed QBF formula $\Phi(\boldsymbol{z}/v)$ resulting from the partial evaluation is true. And we call the truth assignment v a *model* for Φ. Because a subset of free variables determines uniquely a truth assignment, we do not distinguish between truth assignments and subsets of variables.

2.2 Complexity

Next we give a brief review of the relevant notions of complexity theory. P is the class of problems decidable in polynomial time. For the classes Σ_k^P, and Π_k^P of the polynomial hierarchy please see [15].

A problem L is in P/poly (resp. NP/poly) if there is a problem L' in P (resp. NP) and a polynomial $f(n)$ such that $x \in L$ if and only if $(x, f(|x|)) \in L'$. In other words, P/poly (resp. NP/poly) is the class of problems solvable by polynomial–time deterministic (resp. nondeterministic) Turing machines augmented by polynomial advice.

NC^1 is the class of problems solvable by a uniform family of Boolean circuits of logarithmic depth $O(\log n)$ (for detailed definition please see [8]). Similarly, FNC^1 is the of class of functions computable by uniform families of logarithmic depth circuits. NC^1/poly is the analogue of the class NC^1 in which no uniformity conditions on families of circuits are imposed. A classical result on circuit complexity in [23] states that NC^1/poly coincides with the class of languages decidable by polynomial size propositional formulas. NC^1/poly consists of uncountable languages, thus it can not be subset of P. On the other hand, it is strongly believed that P is not a subset of NC^1/poly, either. However, the proof for this conjecture is open.

A problem L in P is called P-complete if any problem L' in P can be NC^1 many-one reduced to L (see [8]). The problem of determining whether a 3HORN formula is unsatisfiable is P-complete.

3 Reductions

In this section we give precise definitions of various reductions. We write $var(T)$ for the set of free variables of a formula T, and $M(T)$ for the set of models of T. Intuitively, a class $\mathcal{C}_1$ of formulas can be model-equivalently reduced to a class $\mathcal{C}_2$ if every formula T in $\mathcal{C}_1$ can be transformed into a theory T' in $\mathcal{C}_2$ such that T and T' have the same number of models, and models of T' can be computed efficiently from those of T. If the transformation costs polynomial time (resp. polynomial space) then we will call the reduction a polynomial-time (resp. polynomial-space) model-equivalent reduction.

Definition 1. *Let $\mathcal{C}, \mathcal{C}'$ be two classes of formulas. We say $\mathcal{C}$ can be* **polynomial–time model–equivalently reduced to** *$\mathcal{C}'$, denoted as $\mathcal{C} \preceq_{ptime} \mathcal{C}'$, if there are two polynomials $p(n)$ and $q(n)$, a $p(n)$-time computable function $f : \mathcal{C} \longrightarrow \mathcal{C}'$, and a $q(n)$-time computable mapping*

$$g : \{(T, v) : T \in \mathcal{C}, v \subseteq var(T)\} \longrightarrow \{u : u \subseteq var(T') \textit{ for some } T' \in \mathcal{C}'\}$$

such that for any fixed $T \in \mathcal{C}$, the mapping g_T, defined by $g_T(v) := g(T, v)$ is a bijection from $M(T)$ to $M(f(T))$.

If the mapping g satisfies $g(T, v) = v$ for all $T \in \mathcal{C}$ and $v \subseteq var(T)$,i.e., T and $f(T)$ are equivalent, then we say $\mathcal{C}$ can be polynomial–time equivalently reduced to $\mathcal{C}'$, denoted as $\mathcal{C} \preceq^{equ}_{ptime} \mathcal{C}'$.

Please note that in the above definition we only know the transformation f and the mapping g, we do not know whether T or $f(T)$ has models. More precisely, for $v \subseteq var(T)$ and $u \subseteq var(f(T))$, we have the following:

(1) if v is a model of T then $g_T(v)$ must be a model of $f(T)$, and
(2) if u is a model of $f(T)$ then $g_T^{-1}(u)$ must be a model of T.

Roughly speaking, that $\mathcal{C}$ can be polynomial-time model-equivalently reduced to $\mathcal{C}'$ means that the representability of $\mathcal{C}'$ is not weaker than that of $\mathcal{C}$ and theories in $\mathcal{C}'$ can be computed efficiently from those in $\mathcal{C}$, in other words, any knowledge base represented as a theory in $\mathcal{C}$ can be transformed efficiently to a theory in $\mathcal{C}'$ such that they have the same abilities. The relations $\preceq_{ptime}$ and $\preceq^{equ}_{ptime}$ are reflexive and transitive.

Suppose the satisfiability problem for $\mathcal{C}_1$ is complete for a level in the polynomial hierarchy while the satisfiability problem for $\mathcal{C}_2$ is at a lower level, then it would be impossible that $\mathcal{C}_1 \preceq_{ptime} \mathcal{C}_2$ unless the polynomial hierarchy collapses. The question is: if $\mathcal{C}_i$ and $\mathcal{C}_j$ have the same complexity up to a polynomial, whether $\mathcal{C}_i$ can be model-equivalently reduced to $\mathcal{C}_j$.

Now we introduce a weaker reduction which only requires that the transformation $f : \mathcal{C} \longrightarrow \mathcal{C}'$ must be computable in polynomial space instead of polynomial time. That means we can spend more time but the length of the calculated formulas are bounded by a polynomial.

Definition 2. *Let $\mathcal{C}$,$\mathcal{C}'$ be two classes of formulas. We say $\mathcal{C}$ can be* **polynomial–space model-equivalently reduced to** *$\mathcal{C}'$, denoted as $\mathcal{C} \preceq_{pspace} \mathcal{C}'$, if there are two polynomials $p(n)$ and $q(n)$, a $p(n)$-space computable function $f : \mathcal{C} \longrightarrow \mathcal{C}'$, and a $q(n)$-time computable mapping*

$$g : \{(T, v) : T \in \mathcal{C}, v \subseteq var(T)\} \longrightarrow \{u : u \subseteq var(T') \text{ for some } T' \in \mathcal{C}'\}$$

such that for any fixed $T \in \mathcal{C}$, the mapping g_T, defined by $g_T(v) := g(T, v)$ is a bijection from $M(T)$ to $M(f(T))$.

If the mapping g satisfies $g(T, v) = v$ for all $T \in \mathcal{C}$ and $v \subseteq var(T)$, i.e., T and $f(T)$ are equivalent, then we say $\mathcal{C}$ can be polynomial–space equivalently reduced to $\mathcal{C}'$, denoted as $\mathcal{C} \preceq^{equ}_{pspace} \mathcal{C}'$.

Roughly speaking, that $\mathcal{C}$ can be polynomial–space model-equivalently reduced to $\mathcal{C}'$ means that the representability of $\mathcal{C}'$ is not weaker than that of $\mathcal{C}$, in other words, any knowledge base represented as a theory T in $\mathcal{C}$ can be represented as a theory T' in $\mathcal{C}'$ such that up to a polynomial they have the same size. However, T' may be not polynomial-time computable. Again, the polynomial–space model–equivalent reduction is reflexive and transitive. Furthermore, any polynomial–time reduction is a polynomial–space reduction.

A *monotone* and *symmetric* formula is a CNF formula φ in which each clause either contains only positive literal or only negative literals and for each clause $(L_1 \vee \cdots \vee L_k) \in \varphi$ the clause $(\neg L_1 \vee \cdots \vee \neg L_k)$ also occurs in φ. Let S-M-CNF be the class of all monotone and symmetric CNF formulas. Please note that any satisfiable formula in S-M-CNF has an even number of satisfying truth assignments. Since there are CNF formulas with an odd number of satisfying truth assignments, we obtain CNF $\not\preceq_{pspace}$ S-M-CNF.

Definition 3. *We write $\mathcal{C}_1 \asymp_{ptime} \mathcal{C}_2$ if $\mathcal{C}_1 \preceq_{ptime} \mathcal{C}_2$ and $\mathcal{C}_2 \preceq_{ptime} \mathcal{C}_1$.*

Likewise for the definition of $\asymp^{equ}_{ptime}$, $\asymp_{pspace}$, and $\asymp^{equ}_{pspace}$.

4 Propositional Formulas and Quantified Boolean Formulas

Every propositional formula can be transformed into a logically equivalent CNF formula. But the problem is the length of the resulting formulas. It is well–known that there are DNF formulas φ_n of length $2n$ for which any equivalent CNF formula has length greater or equal than 2^n. Hence, we obtain DNF$\not\preceq^{equ}_{pspace}$CNF.

If we demand that there is a one-to-one mapping between the models instead of the logical equivalence, then such a reduction exists.

Lemma 1. *DNF $\preceq_{ptime}$ CNF*

Proof. Suppose $\varphi = D_1 \vee D_2 \vee \cdots \vee D_n$ is an arbitrary DNF formula with $D_i = (L_1^i \wedge \cdots \wedge L_{k_i}^i)$. Pick new variables $y_1, \cdots, y_n$. For simplicity, we write φ_i' for the formula $D_i \vee D_{i+1} \vee \cdots \vee D_n \vee y_1 \vee \cdots \vee y_{i-1}$, and write φ_i'' for $D_{i+1} \vee \cdots \vee D_n \vee y_1 \vee \cdots \vee y_{i-1}$. We shall adopt the following procedure to transform φ to a CNF formula which has the same models as φ with respect to a one-to-one mapping. At first, define ψ_1 to be

$$\psi_1 := (L_1^1 \vee \neg y_1) \wedge \cdots \wedge (L_{k_1}^1 \vee \neg y_1) \wedge (\neg(D_2 \vee \cdots \vee D_n) \vee \neg y_1) \wedge \varphi_2'$$

It is easy to see that there is a 1-1 correspondence between models of φ and ψ_1. Further, please note that $\neg(D_2 \vee \cdots \vee D_n) \vee \neg y_1$ can be easily transformed into an equivalent CNF formula by using the distributivity law. Thus ψ_1 is in fact the conjunction of a CNF formula and a DNF formula.

Suppose $\psi_i = \psi_i' \wedge \varphi_{i+1}'$ has been obtained such that ψ_i' is a CNF formula and that ψ_i and φ have the same models with respect to a one-to-one mapping. Now define

$$\psi_{i+1} := \psi_i' \wedge (L_1^{i+1} \vee \neg y_{i+1}) \wedge \cdots \wedge (L_{k_{i+1}}^{i+1} \vee \neg y_{i+1}) \wedge (\neg\varphi_{i+1}'' \vee \neg y_{i+1}) \wedge \varphi_{i+2}'.$$

Finally, ψ_n is a CNF formula which has the same models with φ with respect to a 1-1 correspondence. It is not hard to see that this procedure cost no more than quadratic time. The mapping $g(\varphi, v)$ can be computed in the following way. If v makes $(D_2 \vee \cdots \vee D_n)$ false then define $v_1 := v \cup \{y_1\}$, otherwise, $v_1 := v$. Then define v_2 according to whether v_1 makes $(D_3 \vee \cdots \vee D_n \vee y_1)$ false or not. Continuing this procedure, finally we get v_n. Define $g(\varphi, v) := v_n$. ■

Remark. Please notice that, assuming P $\neq$ NP, CNF cannot be polynomial-time model-equivalently reduced to DNF, CNF$\not\preceq_{ptime}$ DNF, because the SAT–problem for DNF is solvable in linear time whereas the SAT–problem for CNF is NP–complete. However, it is open whether CNF$\preceq_{pspace}$ DNF. Obviously, we can calculate in PSPACE the number of satisfying truth assignments for a CNF formula and construct a DNF formula with the same number of satisfying truth assignments. But the problem is to establish a one-to-one mapping between the models, which can be computed in polynomial time.

Lemma 2. *CNF $\asymp_{ptime}$ 3CNF*

Proof. Clearly, 3CNF $\preceq_{ptime}$ CNF. For the inverse direction we present a procedure to reduce the length of the clauses. For any clause $(L_1 \vee L_2 \vee \cdots \vee L_{k-1} \vee L_k)$ with $k > 3$ we introduce a new variable x, and replace the clause by the following clauses

$$(L_1 \vee L_2 \vee \cdots \vee L_{k-2} \vee \neg x) \wedge (L_{k-1} \vee L_k \vee x) \wedge (L_{k-1} \rightarrow \neg x) \wedge (L_k \rightarrow \neg x).$$

It is easy to see that there is a polynomial–time computable one-to-one mapping between the models of the clause and the models of the associated clauses. Now we apply the procedure as long as clauses of length greater than 3 exists. The length of the generated formula is linear in the length of the initial formula, the procedure can be performed altogether in polynomial time, and finally there is a polynomial–time one–to–one mapping between the models. ■

However, whether PF can be polynomial–time model-equivalently reduced to CNF remains open.

Let us recall the key idea of Tseitin's procedure for transforming a propositional formula into a satisfiability equivalent CNF formula, which is based on the replacement of $\alpha \vee (\beta \wedge \gamma)$ by $(\alpha \vee \neg y) \wedge (\gamma \vee y) \wedge (\beta \vee y)$ for a new variable y (see e.g. [24]). Because of the De Morgan's law we only consider propositional formulas in negation normal form. By the commutative law and the associative law every propositional formula can be written as the conjunction of some formulas in CNF or with the form $\alpha \vee (\beta \wedge \gamma)$. Suppose $\varphi = \varphi' \wedge (\alpha \vee (\beta \wedge \gamma))$ is a such formula, then we obtain that φ and $\exists y(\varphi' \wedge (\alpha \vee \neg y) \wedge (\gamma \vee y) \wedge (\beta \vee y))$ have the same models. By iterative applications of this substitution we can find for any propositional formula φ a CNF formula ψ such that φ and $\exists y_1 \cdots \exists y_m \psi$ have the same models.

Suppose $\Phi = \exists y_1 \cdots \exists y_m(\alpha_1 \wedge \cdots \wedge \alpha_r)$ is in $\exists$CNF*. We can further transform the formula into a formula in $\exists$3CNF*. For each clause $\alpha_i = (L_{i,1} \vee \cdots \vee L_{i,k_i})$ with $k_i > 3$, define

$$\begin{aligned}\varphi(\alpha_i) := {} & (L_{i,1} \vee L_{i,2} \vee \neg y_{i,2}) \\ & \wedge (y_{i,2} \vee L_{i,3} \vee \neg y_{i,3}) \wedge \cdots \wedge (y_{i,(k_i-3)} \vee L_{i,(k_i-2)} \vee \neg y_{i,k_i-2}) \\ & \wedge (y_{i,(k_i-2)} \vee L_{i,(k_i-1)} \vee L_{i,k_i})\end{aligned}$$

where $y_{i,2}, \cdots, y_{i,(k_i-2)}$ are new variable. Clearly,

$$\exists y_1 \cdots \exists y_m \exists y_{i,2} \cdots \exists y_{i,(r_i-2)}(\alpha_1 \wedge \cdots \wedge \alpha_{i-1} \wedge \varphi(\alpha_i) \wedge \alpha_{i+1} \wedge \cdots \wedge \alpha_r)$$

and Φ have the same models (see [13]). Therefore, after several applications finally we can obtain a formula Ψ in $\exists$3CNF* such that Φ and Ψ have the same models. Therefore

Lemma 3. *PF*$\preceq^{equ}_{ptime}$ $\exists$*CNF** $\preceq^{equ}_{ptime}$ $\exists$*3CNF**.

Next we will show that every formula in $\exists$CNF* can be transformed into a propositional formula, but the transformation from $\exists$CNF* into PF involves a significant increase in size.

Suppose $\Phi = \exists y_1 \cdots \exists y_m \varphi$ is a formula with φ in CNF. Let

$$\varphi := \varphi' \wedge (y_1 \vee f_1) \wedge \cdots \wedge (y_1 \vee f_s) \wedge (\neg y_1 \vee g_1) \wedge \cdots \wedge (\neg y_1 \vee g_t),$$

where φ' does not contain any occurrences of y_1 or $\neg y_1$. Define

$$\varphi_1 := \varphi' \wedge \left(\bigwedge_{1 \le i \le s, 1 \le j \le t} (f_i \vee g_j) \right).$$

Lemma 4. *Φ and $\Phi_1 := \exists y_2 \cdots \exists y_m \varphi_1$ have the same models.*

Proof. Suppose v is a truth assignment for which Φ is true. Then we can extend v to v' with $v' = v \cup \{y_1, \cdots, y_m\}$ so that v' satisfies φ. Let v_1 be the restriction of v' to $v \cup \{y_2, \cdots, y_m\}$. If v' makes y_1 false then for v_1, $f_1, \cdots, f_s$ must be true; else $g_1, \cdots, g_t$ are true. Thus, v_1 satisfies φ_1. That is, v is a satisfying truth assignment for Φ_1.

Conversely, suppose v is a truth assignment for Φ_1. Then v can be extended to v_1 with $v_1 = v \cup \{y_2, \cdots, y_m\}$ so that v_1 satisfies φ_1. Suppose without loss of generality that v_1 makes some f_i false. Thus v_1 must make each g_j true. Then we can extend v_1 to v' by setting $y_1 = 1$. Clearly, v' satisfies φ. Thus, v is a satisfying truth assignment for Φ. ■

We may call the procedure from Φ to Φ_1 the complete resolution over y_1. From Lemma 4, successively applying the complete resolution over $y_1, y_2, \cdots, y_m$, we finally get a CNF formula which has the same models with Φ. However, the following example shows that this procedure may costs super-polynomial time.

Example 1. For each number $n > 1$, let

$$\Phi_n := \exists y_1 \cdots \exists y_n \left((\neg y_1 \vee \cdots \vee \neg y_n) \wedge \bigwedge_{1 \le i,j \le n} (\neg x_{i,j} \vee y_i) \right).$$

The length of Φ_n is clearly $O(n^2)$. Applying complete resolution we get a CNF formula

$$\varphi_n := \bigwedge_{1 \le j_1, \cdots, j_n \le n} (\neg x_{1,j_1} \vee \cdots \vee \neg x_{n,j_n}).$$

The length of φ_n is $O(n^n)$. It follows that the CNF formulas generated by the transformation from $\exists$CNF* into CNF may be significantly larger than the original formulas.

However, Φ_n is logically equivalent to

$$\bigvee_{1 \le i \le n} (\neg x_{i,1} \wedge \cdots \wedge \neg x_{i,n})$$

which has length $O(n^2)$. The question is whether any formula in $\exists$CNF* can be polynomial–time model-equivalently reduced to a propositional formula.

Theorem 1. *(1) $\exists$CNF* $\preceq_{ptime}$ PF implies NP=P*
(2) NP$\not\subseteq$P/poly implies $\exists$CNF $\not\preceq_{pspace}$PF*

Proof. (1) Please note that for a formula $\exists \boldsymbol{y} \varphi(x, \boldsymbol{y})$ and a truth assignment for the free variable x, the problem of deciding whether v satisfies $\exists \boldsymbol{y} \varphi(x, \boldsymbol{y})$ (i.e., whether $\exists \boldsymbol{y} \varphi(x/v, \boldsymbol{y})$ is true) is NP-complete. The membership in NP is trivial. The hardness can be seen from the following reduction. For any propositional formula φ over $\boldsymbol{y} = y_1, \cdots, y_m$, it is satisfiable if and only if $\exists \boldsymbol{y}(\varphi \wedge x)$ is true for the truth assignment $x = 1$.

Suppose $\exists$CNF* $\preceq_{ptime}$ PF. Then for any formula $\Phi = \exists \boldsymbol{y}\varphi(x)$ and any truth assignment v for the free variable x, we can construct in polynomial time a propositional formula ψ and a truth assignment v' defined on variables of ψ such that v satisfies Φ if and only if v' satisfies ψ. Since we have only two truth assignments $v(x = 1$ and $x = 0)$, the satisfiability of ψ can be decided in polynomial time. Please note that whether v' satisfies ψ can be solved in polynomial time. Therefore, we obtain NP=P.

(2) For each $n > 0$, let S_n be the set of all formulas in 3CNF with $|\varphi| = n$ and $var(\varphi) \subseteq \{x_1, \cdots, x_n\}$. Define $S := \bigcup_{n>0} S_n$. It is well-known that the satisfiability problem for S is NP-complete.

Let $\pi(n)$ be the set of all 3-clauses over $x_1, \cdots, x_n$. Clearly, the number of clauses in $\pi(n)$ is $O(n^3)$.

We associate with each clause $\gamma \in \pi(n)\}$ a new variable c_γ and we denote $C = \{c_\gamma \mid \gamma \in \pi(n)\}$.

Define

$$T_n := \left(\bigwedge\nolimits_{\gamma\in\pi(n)}(\gamma \vee c_\gamma)\right)$$
$$\Psi_n := \exists x_1 \cdots \exists x_n T_n.$$

Let φ be a 3CNF formula with $|\varphi| = n$. W.l.o.g we assume $var(\varphi) \subseteq \{x_1, \cdots, x_n\}$. Suppose $\varphi = \alpha_1 \wedge \cdots \wedge \alpha_h$. Define a truth assignment v_φ by setting each c_{α_i} to 0, $i = 1, \cdots, h$ and other variables in C to 1. Clearly v_φ can be computed in time polynomial in $|\varphi|$.

Now, it is not hard to see that φ is satisfiable if and only if v_φ satisfies Ψ_n.

Suppose $\exists$CNF* $\preceq_{pspace}$ PF, then there is a sequence $\psi_1, \psi_2, \cdots, \psi_n, \cdots$ of propositional formulas such that

(1) the size of each ψ_n is bounded by a polynomial, and

(2) for each n, there is a polynomial-time computable one-to-one mapping between the models of Ψ_n and ψ_n.

Then we define an advice-taking Turing machine in the following way. The advice oracle is ψ_n. Given an instance φ of S with $|\varphi| = n$, the machine loads ψ_n, then computes v_φ in polynomial time in n, then computes v'_φ according to the one-to-one mapping, finally checks whether v'_φ satisfies ψ_n. Please note that to check whether a truth assignment satisfies a propositional formula can be done in polynomial time. Therefore, the advice-taking machine works in time polynomial in $|\varphi|$. Since the satisfiability problem for S is NP-complete, it follows that NP$\subseteq$P/poly. ■

Theorem 2. *P$\not\subseteq$ NC^1/poly implies $\forall\neg$HORN* $\not\preceq^{equ}_{pspace}$PF.*

Proof. We know that the problem of deciding whether a 3-Horn formulas is unsatisfiable is P-complete.

For each $n > 0$, let S_n be the set of all formulas in 3-Horn with $|\varphi| = n$ and $var(\varphi) \subseteq \{x_1, \cdots, x_n\}$. Define $S := \bigcup_{n>0} S_n$. S is P-complete.

Let $\pi(n)$ be the set of all 3-Horn clauses over $x_1, \cdots, x_n$. Clearly, the number of clauses in $\pi(n)$ is not more than $O(n^3)$.

We associate with each clause $\gamma \in \pi(n)$ a new variable c_γ and we denote $C = \{c_\gamma \mid \gamma \in \pi(n)\}$.

Define

$$T_n := \left(\bigwedge\nolimits_{\gamma\in\pi(n)}(\gamma \vee \neg c_\gamma)\right).$$
$$\Psi_n := \forall x_1 \cdots \forall x_n \neg T_n.$$

Note that the size of Ψ_n is $O(n^3)$, and T_n is a 4-Horn formula.

Let φ be a 3-Horn formula with $|\varphi| = n$. W.l.o.g. we assume $var(\varphi) \subseteq \{x_1, \cdots, x_n\}$. Suppose $\varphi = \alpha_1 \wedge \cdots \wedge \alpha_h$. Define a truth assignment v_φ by setting each c_{α_i} to 1, $i = 1, \cdots, h$ and other variables in C to 0.

Now it is easy to show that φ is unsatisfiable if and only if v_φ satisfies Ψ_n.

Suppose $\forall\neg$HORN* $\preceq^{equ}_{pspace}$ PF. Then there is a sequence of propositional formulas $F_1, F_2, \cdots, F_n, \cdots$ such that

(1) for each n, Ψ_n and F_n have the same models
(2) the size of the formulas F_n are bounded by a polynomial in n.

Then by Spira's theorem [23] which states that NC^1/poly is equal to the class of languages decidable by polynomial size propositional formulas, we obtain that $S \in$NC^1/poly. Since S is P-complete, it follows that P$\subseteq$NC^1/poly. ■

We think that the above theorem is interesting because the satisfiable problem for $\forall\neg$HORN* is solvable in polynomial time, whereas the satisfiability problem for PF is NP-complete. However, for Q2CNF* we have the following.

Proposition 1. *Q2CNF** $\asymp^{equ}_{ptime}$ *2CNF.*

Proof. See Theorem 7.4.6 and Theorem 7.6.1 in [13]. ■

5 Some Classes of Quantified Boolean Formulas

In this section we are interested in the model-equivalent relations between the following classes of quantified Boolean formulas: QCNF*, Q¬CNF*, QHORN*, and

gQEHORN*: the class of formulas $Q(\alpha_1 \wedge \cdots \wedge \alpha_m)$ in QCNF* for which each α_i is a Horn clause after removing the universal literals and free literals.

QEHORN*: the class of formulas $Q(\alpha_1 \wedge \cdots \wedge \alpha_m)$ in QCNF* for which each α_i is a Horn clause after removing the universal literals.

Besides QHORN* all the classes have a satisfiability problem which is PSPACE-complete.

Let $\Phi(\boldsymbol{z}) = Q\varphi(\boldsymbol{z})$, v a truth assignment of $\boldsymbol{z}$. Then $Q\varphi[\boldsymbol{z}/v]$ is true if and only if $\overline{Q}\neg\varphi[\boldsymbol{z}/v]$ is false. Here the prefix $\overline{Q}$ is obtained from Q by replacing existential resp. universal quantifiers by universal resp. existential quantifiers.

Lemma 5. *(1) QBF** $\asymp^{equ}_{ptime}$ *QCNF*.*
*(2) QBF** $\asymp^{equ}_{ptime}$ *Q¬CNF*.*

Proof. (1) QCNF* $\preceq^{equ}_{ptime}$ QBF* is trivial. The other direction follows immediately by means of Lemma 3, since any propositional kernel of a QBF* formula can be transformed into a ∃QCNF* formula in polynomial time preserving the models.

(2) Q¬CNF* $\preceq^{equ}_{ptime}$ QBF* is trivial. For the inverse, let $\Phi = Q\varphi$ be any formula in QBF*. Consider $\overline{Q}\neg\varphi$. By (1), we can find in polynomial time a CNF formula φ' such that $\overline{Q}\neg\varphi$ and $Q'\varphi'$ have the same models. Therefore, $Q\varphi$ and $\overline{Q'}\neg\varphi'$ have the same models. ■

Theorem 3. *(1) QBF** $\asymp^{equ}_{ptime}$ *QCNF** $\asymp^{equ}_{ptime}$ *gQEHORN*.*
*(2) QCNF** $\not\preceq^{equ}_{pspace}$ *QEHORN*.*
(3) Suppose $\Sigma_2^P \neq NP$*. Then for any fixed natural number* $k \geq 1$*,* ∃*CNF** $\not\preceq_{ptime}$ Σ_k*-gQEHORN*.*

Proof. (1) We only need to show QCNF* $\preceq^{equ}_{ptime}$ gQEHORN*. In [7], a reduction is given which transforms each formula Φ in QCNF into a formula $\mathcal{F}(\Phi)$ in QEHORN such that the formulas have the same truth value (see Theorem 6.33 in [7]). We just use the same reduction except allowing free variables.

Let $\Phi(\boldsymbol{z}) = \forall\boldsymbol{w}_1\exists\boldsymbol{x}_1\cdots\boldsymbol{w}_k\exists\boldsymbol{x}_k(\varphi_1\wedge\cdots\wedge\varphi_m)$ with $\boldsymbol{w}_i = w_{s_{i-1}+1},\cdots,w_{s_i}$ and $\boldsymbol{x}_i = x_{t_{i-1}+1},\cdots,x_{t_i}$ for $i = 1,\cdots,k$, and $\boldsymbol{z} = z_1,\cdots,z_n$. Here s_1 and n are possible to be 0.

Pick new variables $y_0, y_1, \cdots, y_{t_k}$ and y'_{t_k} which do not occur in Φ.

Please note that every clause φ_i consists of three parts: some free literals, some universal literals, and some existential literals. $[\varphi_i]$ is the clause obtained from φ_i by replacing each existential literal by its negation.

Now define Φ' to be the following formula:

$$
\begin{array}{l}
\Phi'(\boldsymbol{z}) = \\
\qquad \forall\boldsymbol{w}_1\exists y_0\forall x_1\exists y_1\forall x_2\cdots\exists y_{t_1-1}\forall x_{t_1} \\
\qquad \forall\boldsymbol{w}_2\exists y_{t_1}\forall x_{t_1+1}\exists y_{t_1+1}\forall x_{t_1+2}\cdots\exists y_{t_2-1}\forall x_{t_2} \\
\qquad \vdots \\
\qquad \forall\boldsymbol{w}_k\exists y_{t_{k-1}}\forall x_{t_{k-1}+1}\exists y_{t_{k-1}+1}\forall x_{t_{k-1}+2}\cdots\exists y_{t_k-1}\forall x_{t_k}\exists y_{t_k}\exists y'_{t_k} \\
\qquad \left(\begin{array}{ll}
(x_1\vee\cdots\vee x_{t_k}\vee y_{t_k}) & \wedge \\
(\neg x_1\vee\cdots\neg x_{t_k}\vee\neg y'_{t_k}) & \wedge \\
\bigwedge_{j=0}^{t_k-2}(y'_{t_k}\rightarrow x_{j+1}\vee\neg x_{j+2}\vee\cdots\vee\neg x_{t_k}\vee y_j) & \wedge \\
\bigwedge_{j=0}^{t_k-2}(y'_j\rightarrow\neg x_{j+1}\vee x_{j+2}\vee\cdots\vee x_{t_k}\vee y_{t_k}) & \wedge \\
(y'_{t_k}\rightarrow x_{t_k}\vee y_{t_k-1}) & \wedge \\
(y_{t_k-1}\rightarrow\neg x_{t_k}\vee y_{t_k}) & \wedge \\
(y_{t_k}\rightarrow[\varphi_1)]\vee y'_{t_k}) & \wedge \\
\vdots & \\
(y_{t_k}\rightarrow[\varphi_m]\vee y'_{t_k}) &
\end{array}\right)
\end{array}
$$

Clearly, Φ' is in gQEHORN*. It is easy to see that $\Phi'(\boldsymbol{z}/v)$ is $\mathcal{F}(\Phi(\boldsymbol{z}/v))$ for each truth assignment v on $\boldsymbol{z}$. Thus we have

$$\Phi(\boldsymbol{z}/v) \text{ is true if and only if } \mathcal{F}(\Phi(\boldsymbol{z}/v)) \text{ is true if and only if } \Phi'(\boldsymbol{z}/v) \text{ is true.}$$

(2) By Theorem 7.4.6 in [13] we know that every formula in QEHORN* is equivalent to a Horn formula. QCNF* $\preceq^{equ}_{pspace}$ QEHORN* would imply that every QCNF* formula, especially, every CNF formula is logically equivalent to a Horn formula. But the CNF formula $(x_1 \vee x_2)$ has no logically equivalent Horn formula.

(3) Suppose $\exists$CNF* $\preceq_{ptime}$ Σ_k-gQEHORN*. Then for any formula $\Phi(\boldsymbol{z})$ in $\exists$CNF*, we can find in polynomial time a formula $\Psi(\boldsymbol{z'})$ in Σ_k-gQEHORN* such that Φ and Ψ have the same models wrt a one-to-one mapping. Then $\forall \boldsymbol{z}\Phi$ is false if and only if $\forall \boldsymbol{z'}\Psi$ is false. Please notice that $\forall \boldsymbol{z'}\Psi$ is in QEHORN. By Theorem 3.3 in [12] the Π_{k+1}-QEHORN is in co-NP. However, Π_2-QCNF is Π_2^P-complete. The assertion follows. ■

Because it is widely believed that the polynomial hierarchy does not collapse, the expressiveness of QEHORN* is much weaker than QCNF* although they have the same complexity (PSPACE-complete). For gQHORN*, which has the same representability with QCNF*, its subclasses Σ_k-gQEHORN* are even not stronger than Σ_1-QCNF* for any $k \geq 1$.

Theorem 4. *Suppose $\Sigma_2^P \not\subseteq NP/poly$. Then for any fixed natural number $k \geq 1$, Π_2-CNF* $\not\preceq_{pspace}$ Π_k-gQEHORN*.*

Proof. Let S_n be the class of formulas $\forall \boldsymbol{x}_1 \exists \boldsymbol{x}_2 \varphi$ with $\boldsymbol{x}_i = x_{(i-1)n+1}, \cdots, x_{in}$ for $i = 1, 2$, φ is a 3CNF formula over variables $x_1, \cdots, x_{2n}$, and define

$$S := \bigcup_{n=1}^{\infty} S_n.$$

For any Π_2-Q3CNF formula Φ we can add some dummy quantifiers in the prefix and rename the variable properly to get a formula Ψ in S such that Φ and Ψ have the same satisfiability.

Let $\pi(n)$ be the set of all 3-clauses over $x_1, \cdots, x_n, x_{n+1}, \cdots, x_{2n}$. Clearly, the number of clauses in $\pi(n)$ is $O(n^3)$.

We associate with each clause $\gamma \in \pi(n)$ a new variable c_γ and we denote $C = \{c_\gamma \mid \gamma \in \pi(n)\}$.

Define

$$\begin{aligned} T_n &:= \left(\bigwedge\nolimits_{\gamma \in \pi(n)} (\gamma \vee c_\gamma) \right) \\ \Psi_n &:= \forall \boldsymbol{x}_1 \exists \boldsymbol{x}_2 T_n. \end{aligned}$$

Please note that $|\Psi_n|$ is $O(n^3)$. Let $\forall \boldsymbol{x}_1 \exists \boldsymbol{x}_2 \varphi$ be a formula in S_n. Suppose $\varphi = \alpha_1 \wedge \cdots \wedge \alpha_h$. Define a truth assignment v by setting each c_{α_i} to 0, $i = 1, \cdots, h$ and other names in C to 1. Clearly v is computed in time polynomial in $|\varphi|$.

Now we can see that $\forall \boldsymbol{x}_1 \exists \boldsymbol{x}_2 \varphi$ is satisfiable if and only if v satisfies Ψ_n.

Let k be a fixed natural number. Suppose Π_2-QCNF* $\preceq_{pspace}$ Π_k-gQEHORN*. Then there is a Ψ'_n in Π_k-gQEHORN* with polynomial size in n such that there is a in polynomial time computable one-to-one mapping between models of Ψ_n and Ψ'_n. Let C' be the set of free variables in Ψ'_n and v' is the truth assignment corresponding to v. Then $\Psi'_n(C'/v')$ is a Π_k-QEHORN formula. By Theorem 3.3 in [12], the problem of determining the truth of Π_k-QEHORN formulas is in co-NP. The theorem follows. ■

6 Conclusions and Open Questions

In this paper, we have introduced two model-equivalent relations—polynomial-time reduction and polynomial–space reduction between theories, which we think are finer notions to compare the expressiveness of different theories. We have compared according to these notions some classes of propositional formulas and quantified Boolean formulas. Our results show that classes of theories with the same complexity might have different representation strength under some conjectures which are widely believed in computation complexity theory. The main results of this paper are summarized by the following diagram. The existence of a path from class $\mathcal{C}_1$ to class $\mathcal{C}_2$ means that $\mathcal{C}_1$ can be polynomial–time model–equivalently reduced to $\mathcal{C}_2$. Whereas the non-existence of pathes means that $\mathcal{C}_1 \not\preceq_{pspace} \mathcal{C}_2$ or $\mathcal{C}_1 \not\preceq^{equ}_{pspace} \mathcal{C}_2$ (under some conjectures) or it is unknown whether $\mathcal{C}_1 \preceq_{pspace} \mathcal{C}_2$ holds.

$$\begin{tikzcd}
 & \text{3CNF} \arrow[d, shift left] & & \exists\text{3CNF}^* \arrow[d, shift left] & \text{QEHORN}^* \arrow[d] & \Pi_{2k}\text{-QEHORN}^* \arrow[l] \arrow[d] \\
\text{DNF} \arrow[r] & \text{CNF} \arrow[u, shift left] \arrow[r] & \text{PF} \arrow[r] & \exists\text{CNF}^* \arrow[u, shift left] \arrow[r] & \text{QCNF}^* \arrow[dl, shift left] \arrow[dr, shift left] & \Pi_{2k}\text{-gQEHORN}^* \arrow[l] \arrow[d] \\
 & & \text{Q}\neg\text{HORN}^* \arrow[r] & \text{Q}\neg\text{CNF}^* \arrow[ur, shift left] & & \text{gQEHORN}^* \arrow[ul, shift left]
\end{tikzcd}$$

However, this paper is the first step to explore the expressiveness by studying model-equivalence relations. Several issues remain for further work. More technically, we have the following open questions:

1: Does CNF$\preceq_{pspace}$ DNF hold?
2: Does PF $\asymp_{ptime}$ 3CNF hold?
3: Is $\exists$CNF* a $\preceq_{ptime}$-maximal NP-complete class?
4: How about the model-equivalence relation between Π_{2k}-QEHORN* and Π_{2k+2}-QEHORN*. (The satisfiability problem for both classes is in co-NP (see [12]) for any fixed number $k \geq 1$).
5: How about the model-equivalence relation between Π_{2k}-QHORN* and Π_{2k+2}-QHORN*. (The satisfiability problem for both classes is solvable in polynomial time (see [12]) for any fixed number $k \geq 1$). (Just recently, Uwe

Bubeck, Hans Kleine Büning and Xishun Zhao claimed that they solved this question by showing QHORN* $\preceq^{equ}_{ptime}$ $\exists$HORN*.)

References

1. Ben-Eliyahu, R., Dechter, R., Default Logic, Propositional Logic and Constraints, in: *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91),* 1991.
2. Cadoli, M., Donini, F.M., Schaerf, M., Is Intractability of Nonmonotonic Reasoning a Real Drawback, *Artificial intelligence*, 1996, **88**(2): 215-251.
3. K. Clark, Negation as failure, in H. Gallaire and J. Minker (eds.): *Logic and Data Bases*, New York: Plenum Press, 293-322.
4. H.D. Ebbinhaus, J. Flum, *Finite Model Theory*, Springer, 1999.
5. T. Eiter, G. Gottlob, On the Computational Cost of Disjunctive Logic Programming: Propositional Case, *Annals of Mathematics and Artificial Intelligence* **15** (1995), 289-323.
6. F. Fages, Consistency of Clark's Completion and Existence of Stable Models, *Journal of Methods of Logic in Computer Science* **1** (1994), 51-60.
7. A. Flögel, Resolution für Quantifizierte Boole'sche Formeln, Disertation, Paderborn University, 1993.
8. R. Greenlaw, H.J. Hoover, and W.L. Ruzzo, *Limits to Parallep Computation: P-Completeness theory.* Oxford University Press, 1995.
9. M. Gelfond, V. Lifschitz, The Stable Model Semantics for Logic Programming, In *Proceedings of the 5th International Conference on Logic Programming*, 1070-1080, The MIT Press, 1988.
10. M. Gelfond, V. Lifschitz, Classical Negation in Logic Programs and Disjunctive Database, *New Generation Computing*, **9** (1991), 365-385.
11. G. Gogic, H. Kautz, C. Papadimitriou, and B. Selman, *The Comparative Liguistics of Knowledge Representation*, In *Proceedings of IJCAI-95*, Montreal, Canada, 1995.
12. M. Karpinski, H. Kleine Büning, and P.H. Schmitt, On the Computational Complexity of Quantified Horn Clauses, *Lecture Notes in Computer Science* **329**, 129-137, Springer-Verlag, 1987.
13. H. Kleine Büning and T. Lettmann, *Propositional Logic: Deduction and Algorithms*, Cambridge University Press, 1999.
14. H.Kleine Büning, X. Zhao, Equivalence Models for Quantified Boolean Formulas, *SAT2004*, Vancourver, 2004.
15. D.S. Johnson, A Catalog of Complexity Classes, In *Handbook of Theoretical Computer Science* (ed. J. van Leeuwen), Vol. A, 67-161, Elsevier Science Publisher (North-Holland), Amsterdam, 1990.
16. J. Lee, F. Lin: Loop Formulas for Circumscription. *AAAI 2004*: 281-286.
17. J. Lee and V. Lifschitz, Loop Formulas for Disjunctive Logic Programs, *Nineteenth International Conference on Logic Programming (ICLP-03)*, 451-465, Mumbai, India, 2003.
18. V. Lifschitz, A. Razborov, Why Are There So Many Loop Formulas, Manuscription, 2003.
19. V. Lifschitz, D. Pearce, and A. Valverde, Strongly Equivalent Logic Programs, *ACM Transactions on Computational Logic* **2** (2001), 526-541.
20. F. Lin, Y. Zhao, ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers, *Artifficial Intelligence* **157**(2004),115-137.

21. Konolige, K., On the Relation between Default and Autoepistemic Logic, *Artificial Intelligence*, 1988, **35**(3): 343-382.
22. W. Marek, M. Truszczyński, *Nonmonotonic Logic*, Springer-Verlag, Berlin, 1993.
23. P.M. Spira, On Time Hardware Complexity Tradeoffs for Boolean Functions, in *Proceedings of the 4th Hawaii Symposium on System Science*, 525-527, Western Periodicals Company, North Hollywood, 1971.
24. G.S. Tseitin, On the Complexity of Derivation in Propositional Calculus, in A.O. Silenko (ed.): *Studies in Constructive Mathematics and Mathematical Logic*, Part II, 1970, 115-125.

Improved Exact Solvers for Weighted Max-SAT*

Teresa Alsinet[1], Felip Manyà[2], and Jordi Planes[1]

[1] DIEI, Universitat de Lleida, Jaume II,
69, E-25001 Lleida, Spain
[2] IIIA-CSIC, Campus UAB,
08193 Bellaterra, Spain

Abstract. We present two new branch and bound weighted Max-SAT solvers (`Lazy` and `Lazy`*) which incorporate original data structures and inference rules, and a lower bound of better quality.

1 Introduction

In recent years we have seen an increasing interest in propositional satisfiability (SAT) that has led to the development of fast and sophisticated complete SAT solvers. Unfortunately, such solvers are not able to solve Max-SAT: Given a Boolean CNF formula ϕ, find a truth assignment that minimizes the number of unsatisfied clauses in ϕ. A more general and related problem is weighted Max-SAT. In this case, a positive weight is associated with each clause and the problem consists of finding a truth assignment that minimizes the sum of weights of unsatisfied clauses. When all the clauses have at most k literals per clause (weighted) Max-SAT is called (weighted) Max-k-SAT.

In this paper we first present two new branch and bound weighted Max-SAT solvers: `Lazy` and `Lazy`*. They differ from previous solvers in the data structures used to represent and manipulate CNF formulas, in the simplification preprocessing techniques applied, in the lower bound, in the variable selection heuristic (which is static in our solvers), and in the incorporation of the Dominating Unit Clause (DUC) rule. We then report on an experimental investigation we have conducted in order to evaluate our solvers on (weighted) Max-SAT instances. The results obtained provide experimental evidence that our solvers are very competitive and outperform some of the best performing (weighted) Max-SAT solvers on a wide range of instances.

2 Description of Lazy

The space of all possible assignments for a CNF formula ϕ can be represented as a search tree, where internal nodes represent partial assignments and leaf nodes

* Research partially supported by projects TIN2004-07933-C03-03 and TIC2003-00950 funded by the *Ministerio de Educación y Ciencia*. The second author is supported by a grant *Ramón y Cajal*.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 371–377, 2005.

represent complete assignments. A branch and bound algorithm for weighted Max-SAT explores the search tree in a depth-first manner. At every node, the algorithm compares the sum of the weights of the number of clauses unsatisfied by the best complete assignment found so far —called upper bound (UB)— with the sum of the weights of the number of clauses unsatisfied by the current partial assignment ($unsat$) plus an underestimation of the sum of the weights of the number of clauses that become unsatisfied if we extend the current partial assignment into a complete assignment ($underestimation$). The sum $unsat + underestimation$ is called lower bound (LB). Obviously, if $UB \leq LB$, a better assignment cannot be found from this point in search. In that case, the algorithm prunes the subtree below the current node and backtracks to a higher level in the search tree. If $UB > LB$, it extends the current partial assignment by instantiating one more variable; which leads to create two branches from the current branch: the left branch corresponds to instantiate the new variable to false, and the right branch corresponds to instantiate the new variable to true. In that case, the formula associated with the left (right) branch is obtained from the formula of the current node by deleting all the clauses containing the literal $\neg p$ (p) and removing all the occurrences of the literal p ($\neg p$); i.e., the algorithm applies the one-literal rule. The solution to weighted Max-SAT is the value that UB takes after exploring the entire search tree.

Lazy implements a branch and bound solver for weighted Max-SAT that incorporates a number of improvements:

- First upper bound: Before starting to explore the search tree, Lazy obtains an upper bound on the sum of the weights of unsatisfied clauses in an optimal solution using a variant of the local search procedure GSAT [8]. This technique was first used by Borchers & Furman in their solver BF [3].
- Formula reduction preprocessing: Before the search starts, Lazy simplifies the initial formula by applying the resolution rule to a particular class of binary clauses. We replace every pair of clauses $(p_1 \vee p_2, w_1)$ and $(\neg p_1 \vee p_2, w_2)$ with the clauses $(p_2, \min(w_1, w_2)), (p_1 \vee p_2, w_1 - \min(w_1, w_2))$ and $(\neg p_1 \vee p_2, w_2 - \min(w_1, w_2))$.[1] The advantage of this preprocessing is that we generate new unit clauses.
- Boolean Constraint propagation: When branching is done, branch and bound algorithms for weighted Max-SAT apply the one-literal rule (simplifying with the branching literal) instead of applying unit propagation as in most SAT solvers.[2] If unit propagation is applied at each node, the algorithm can return a non-optimal solution. However, whenever a unit clause has a weight greater than or equal to the difference between the lower bound and the upper bound, unit propagation can be safely applied. This technique, which Lazy incorporates, was also first used by Borchers & Furman [3].

[1] Clauses with weight 0 are removed from the formula. Note that when $w_1 = w_2 = 1$, which corresponds to Max-SAT, only $(p_2, 1)$ is added.

[2] By unit propagation we mean the repeated application of the one-literal rule until a saturation state is reached.

- Lower bound: $\texttt{LB}_{\text{Lazy}}(\phi) = unsat + \sum_{p \text{ occurs in } \phi} min(ic(p), ic(\neg p))$, where ϕ is the formula associated with the current partial assignment, and $ic(p)$ ($ic(\neg p)$) —inconsistency count of p ($\neg p$)— is the sum of the weights of the clauses that become unsatisfied if the current partial assignment is extended by fixing p to true (false). Note that $ic(p)$ ($ic(\neg p)$) coincides with the sum of the weights of unit clauses of ϕ that contain $\neg p$ (p). Such a lower bound is a generalization to weighted Max-SAT of the lower bound for Max-SAT defined in [10].
- Dominating Unit Clause (DUC) rule: DUC [6] is an inference rule that allows one to fix the truth value of a variable; i.e., it avoids to apply branching on that variable. We have generalized DUC to weighted Max-SAT as follows: If the sum of the weights of the clauses in which appears a literal p ($\neg p$) is not bigger than the sum of the weights of the unit clauses in which appears the literal $\neg p$ (p), then the value of p can be set to false (true). `Lazy` only applies this rule when the input formula contains at most two literals per clause.
- Data structures: Existing (weighted) Max-SAT solvers use adjacency lists to represent formulas, and their variable selection heuristics are typically dynamic. `Lazy` uses a static variable selection heuristic (defined bellow) that allows us to implement extremely efficient data structures for representing and manipulating formulas. Our data structures take into account the following fact: we are only interested in knowing when a clause has become unit or empty. Thus, if we have a clause with four variables, we do not perform any operation in that clause until three of the variables appearing in the clause have been instantiated; i.e., we delay the evaluation of a clause with k variables until $k - 1$ variables have been instantiated. Each clause is represented by an ordered list, which contains the literals of the clause ordered following the order used to instantiate variables. As we instantiate the variables using a static order, we do not have to evaluate a clause until the variable of the penultimate literal in the clause has been instantiated.

  The data structures are defined as follows: Clauses are represented by ordered lists, and we have a pointer to the penultimate literal and to the last literal of the clause. When a variable p is fixed to true (false), the clauses whose penultimate literal is $\neg p$ (p) are evaluated. If some of the instantiated literals in the clause is true, the clause become satisfied; otherwise, we derive a unit clause with the same weight whose only literal is the last literal of the clause. This approach has two advantages: the cost of maintaining that data structure when the solvers backtracks is constant (we do not have to undo pointers like in adjacency lists) and, at each step, we evaluate a minimum number of clauses (we do not evaluate all the clauses that contain the variable we are instantiating, we only evaluate the clauses in which the penultimate literal contains that variable). In addition, we also maintain an array that contains, for each literal, the sum of the weights of the unit clauses in which that literal appears. This array is used to derive empty clauses and to compute lower bounds. This array is the only data structure that the algorithm maintains when it backtracks.

– Static variable selection heuristic: Lazy uses heuristic MOMS [7] adapted to weighted Max-SAT. Lazy orders the variables by the sum of the weights associated with the clauses of minimum size in which the variable appears. Variables are instantiated following that ordering.

3 Description of Lazy*

Lazy* is basically Lazy without applying DUC and with a lower bound of better quality. The new lower bound, $\mathrm{LB}_{\mathrm{Lazy}^\star}$, can be understood as $\mathrm{LB}_{\mathrm{Lazy}}$ extended with a specialization of the so-called star rule [6] that we have generalized to weighted Max-SAT. The star rule for Max-SAT states that if we have a clause of the form $l_1 \vee \cdots \vee l_k$, where $l_1, \ldots, l_k$ are literals, and k unit clauses of the form $\neg l_1, \ldots, \neg l_k$, then the lower bound can be incremented by one. In our case, we only consider clauses of length two, [3] and it is defined as follows: if we have a binary clause $(l_1 \vee l_2, w_1)$ and two unit clauses $(\neg l_1, w_2)$ and $(\neg l_2, w_3)$, then we can increment the lower bound by $w = \min(w_1, w_2, w_3)$ and replace those clauses with $(l_1 \vee l_2, w_1 - w), (\neg l_1, w_2 - w)$, and $(\neg l_2, w_3 - w)$.[4]

The pseudo-code of $\mathrm{LB}_{\mathrm{Lazy}^\star}$, for an input formula ϕ, is defined as follows:[5]

```
 1: $\mathrm{LB}_{\mathrm{Lazy}^\star}(\phi) := 0$
 2: **for** every clause $(l_1 \vee l_2, w_1) \in \phi$ **do**
 3:    **if** $(\neg l_1, w_2) \in \phi$ and $(\neg l_2, w_3) \in \phi$ **then**
 4:       $w := \min(w_1, w_2, w_3)$
 5:       $\mathrm{LB}_{\mathrm{Lazy}^\star}(\phi) := \mathrm{LB}_{\mathrm{Lazy}^\star}(\phi) + w$
 6:       $\phi := \phi - \{(l_1 \vee l_2, w_1), (\neg l_1, w_2), (\neg l_2, w_3)\} \cup$
 7:                 $\{(l_1 \vee l_2, w_1 - w), (\neg l_1, w_2 - w), (\neg l_2, w_3 - w)\}$
 8:    **end if**
 9: **end for**
10: $w' := \mathrm{LB}_{\mathrm{Lazy}^\star}(\phi)$
11: $\mathrm{LB}_{\mathrm{Lazy}^\star}(\phi) := \mathrm{LB}_{\mathrm{Lazy}}(\phi) + w'$
```

To compute the new lower bound efficiently we have also modified the data structures of Lazy. We now have pointers, besides to the last and penultimate literals, to the second from last literal of each clause. In addition, we maintain an array that contains, for each literal, the sum of the weights of the unit clauses in which that literal appears, and an array that contains the binary clauses. The idea behind the new data structure is that we keep track of both the unit and binary clauses that are generated as the search proceeds. This way we are able to compute the new lower bound efficiently.

[3] For longer clauses the star rule did not lead to performance improvements in our experimental investigation.

[4] Clauses with weight 0 are removed from the formula. Note that when $w_1 = w_2 = w_3 = 1$, which corresponds to Max-SAT, no clause is added.

[5] Note that after applying our variant of the star rule, Lazy* applies the lower bound of Lazy to the resulting formula (line 11 of the pseudo-code).

4 Experimental Results

We conducted an experimental investigation in order to compare the performance of Lazy and Lazy* with the following state-of-the-art solvers:

- BF [3]: It is a branch and bound weighted Max-SAT solver which uses MOMS as dynamic variable selection heuristic, and neither it considers underestimations in the lower bound nor formula reduction preprocessing and DUC. Formulas are represented using adjacency lists. It was developed by Borchers & Furman in 1999.
- AMP [2]: It is a branch and bound Max-SAT solver based on BF that incorporates the lower bound of Lazy, and uses the Jeroslow-Wang rule [5][6] as dynamic variable selection heuristic. It was developed by Alsinet, Manyà & Planes in 2003.
- GLMS [4]: It is a weighted Max-SAT solver that encodes the input instance as a weighted constraint network and solves that network with a state-of-the-art weighted Max-CSP solver with a sophisticated and good performing lower bound. It was developed by Givry, Larrosa, Meseguer & Schiex in 2003.
- XZ [11]: It is a branch and bound weighted Max-SAT solver developed by Xing & Zhang in 2004. We used the second release of this solver which is known as MaxSolver.
- AGN [1]: It is a branch and bound Max-2-SAT solver (the weighted version was not implemented). It was developed by Alber, Gramm & Niedermeier in 1998.
- ZSM [12]: It is a branch and bound Max-2-SAT solver (the weighted version was not implemented). It was developed by Zhang, Shen & Manyà in 2003. Improved lower bounds for this solver are described in [9].

As benchmarks we used randomly generated (weighted) Max-2-SAT and (weighted) Max-3-SAT instances. The experiments were performed on a 2GHz Pentium IV with 512 Mb of RAM under Linux.

In our first experiment, we generated sets of random Max-2-SAT instances with 50 and 100 variables and a different number of clauses. Each set had 500 instances. The results of solving such instances with BF, AMP, GLMS, ZSM, AGN, Lazy and Lazy* are shown in Fig. 1. Along the horizontal axis is the number of clauses, and along the vertical axis is the mean time (in seconds) needed to solve an instance of a set. Notice that we use a log scale to represent run-time. Observe that Lazy outperforms the rest of solvers in almost all the instances, even ZSM and AGN that are specifically designed to solve Max-2-SAT instances. GLMS shows a good behavior on large clauses/variables ratios.

In our second experiment, we generated sets of random Max-3-SAT instances with 50 variables and a different number of clauses. Each set had 300 instances.

[6] Given a formula ϕ, for each literal l of ϕ the following function is defined: $J(l) = \sum_{l \in C \in \phi} 2^{-|C|}$, where $|C|$ is the length of clause C. It selects a variable p of ϕ among those that maximize $J(p) + J(\neg p)$.

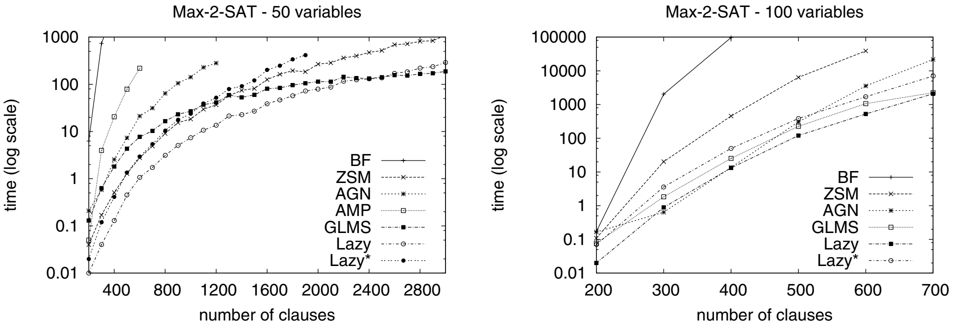


Fig. 1. Experimental results for 50-variable and 100-variable Max-2-SAT instances. Mean time (in seconds)

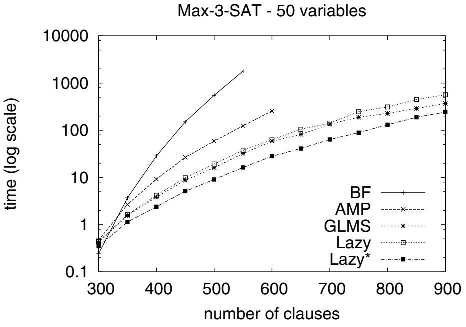


Fig. 2. Experimental results for 50-variable Max-3-SAT instances. Mean time (in seconds)

The results of solving such instances with BF, AMP,GLMS, Lazy and Lazy* are shown in Fig. 2. We observe that Lazy* outperforms the rest of solvers, and that GLMS and Lazy have a similar behavior. BF and AMP are much worse than the rest of solvers. When the input formula is a Max-3-SAT instance, the only difference between Lazy and Lazy* is the lower bound. The results indicate that for Max-3-SAT pays off to use a more sophisticated lower bound.

In our third experiment, we generated sets of random weighted Max-2-SAT and weighted Max-3-SAT instances with 50 variables and a different number of clauses. Each set had 500 instances. The results of solving such instances with BF, AMP, XZ, GLMS, Lazy and Lazy* are shown in Fig. 3. We observe that Lazy is the best performing solver for weighted Max-2-SAT, and Lazy* is the best performing solver for weighted Max-3-SAT. XZ has a good performance profile for weighted Max-2-SAT, but it is not competitive for weighted Max-3-SAT. GLMS is competitive in both cases, while BF and AMP are not competitive. It is worth mentioning that weighted Max-CSP has been intensively studied in

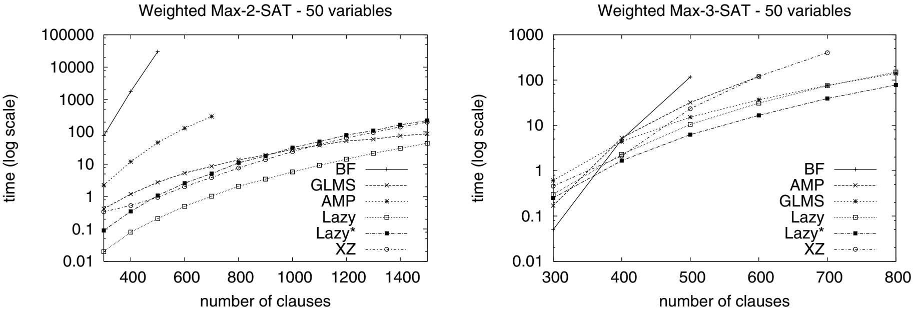


Fig. 3. Experimental results for 50-variable weighted Max-2-SAT and weighted Max-3-SAT instances. Mean time (in seconds)

the constraint programming community during the last decade, while solvers for weighted Max-SAT have only been recently investigated in the SAT community.

References

1. J. Alber, J. Gramm, and R. Niedermeier. Faster exact algorithms for hard problems: A parameterized point of view. In *25th Conf. on Current Trends in Theory and Practice of Informatics*, pages 168–185, 1998.
2. T. Alsinet, F. Manyà, and J. Planes. Improved branch and bound algorithms for Max-SAT. In *Proceedings of SAT-2003*, 2003.
3. B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *J. of Combinatorial Optimization*, 2:299–306, 1999.
4. S. de Givry, J. Larrosa, P. Meseguer, and T. Schiex. Solving Max-SAT as weighted CSP. In *Proceedings of CP-2003*, pages 363–376. Springer LNCS 2833, 2003.
5. R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.
6. R. Niedermeier and P. Rossmanith. New upper bounds for maximum satisfiability. *Journal of Algorithms*, 36:63–88, 2000.
7. D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In *Proceedings of the DIMACS Challenge II Workshop*, 1993.
8. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of AAAI-2002*, pages 440–446, 1992.
9. H. Shen and H. Zhang. Study of lower bound functions for max-2-sat. In *Proceedings of AAAI-2004*, pages 185–190, 2004.
10. R. Wallace and E. Freuder. Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems. In *Cliques, Coloring and Satisfiability*, volume 26, pages 587–615. 1996.
11. Z. Xing and W. Zhang. Efficient strategies for (weighted) maximum satisfiability. In *Proceedings of CP-2004*, pages 690–705, 2004.
12. H. Zhang, H. Shen, and F. Manya. Exact algorithms for MAX-SAT. In *4th Int. Workshop on First order Theorem Proving*, 2003.

Quantifier Trees for QBFs

Marco Benedetti*

Istituto per la Ricerca Scientifica e Tecnologica (IRST),
Via Sommarive 18, 38055 Povo, Trento, Italy
benedetti@itc.it

Abstract. We present a method—called quantifier tree reconstruction—that allows to efficiently recover *ex-post* a portion of the internal structure of QBF instances which was hidden as a consequence of the cast to prenex normal form. Means to profit from a quantifier tree are presented for all the main families of QBF solvers. Experiments on QBFLIB instances are also reported.

1 Introduction

Quantified Boolean Formulas (QBFs) are often represented in the standard *prenex conjunctive normal form*. Standardization enables a complete decoupling between the production of instances and their solution, with countless benefits. On the negative side, one should consider the loose of structural information that happens when instances are flattened onto a fixed format: Experiments show that the original structure of the formula could be exploited to achieve gains in solving performances [13]. We propose an approach that preserves the benefits of standardization and attempts to recover part of the lost information *ex-post*. QBFs are converted to prenex CNF in two steps:

a. A prenex form is generated by moving all the quantifiers outside an (arbitrarily shaped) quantifier-free matrix.

b. The matrix is converted into conjunctive normal form.

One source of structure obfuscation here is that linear prefixes impose unnecessarily strong restrictions w.r.t. the intrinsic dependencies between existential and universal variables in a given matrix. So, we revert Step **a**, to some extent, even if the matrix is left in CNF form (Section 3). In essence, we extract a tree-shaped syntactic structure—which we call *quantifier tree*—out of a flat prenex conjunctive normal form instance. Then, we show (Section 4) how QBF solvers can profit from a quantifier tree (we consider three major families of CNF-based approaches to QBF satisfiability).

Many related works exist, ranging from the *miniscoping* technique in FOL to *quantifier shifting* algorithms for QBFs. We refer the reader to [1] for a comparison with these works, and for an in-depth presentation of the quantifier-tree extraction technique.

* This work is funded by PAT (*Provincia Autonoma di Trento*, Italy), under grant n. 3248/2003.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 378–385, 2005.

2 QBFs and Their Structure

We consider prenex CNF formulas $F = \mathcal{Q}_1 V_1 \mathcal{Q}_2 V_2 \ldots \mathcal{Q}_n V_n \ \widetilde{F}$, where the matrix $\widetilde{F}$ is a CNF on variables $var(F)$, and the prefix $\mathcal{Q}_1 V_1 \mathcal{Q}_2 V_2 \ldots \mathcal{Q}_n V_n$ is such that $\mathcal{Q}_i \in \{\forall, \exists\}, i = 1, \ldots, n$ and $\mathcal{Q}_i \neq \mathcal{Q}_{i+1}, i = 1, \ldots, n-1$, while $\{V_i\}$ is a partition of $var(F)$. A *scope* V_i is *existential* (*universal*) if $\mathcal{Q}_i = \exists$ ($\mathcal{Q}_i = \forall$). The scope $\sigma(v)$ of a variable is the index i such that $v \in V_i$. Variables $v \in V_i$ are existentially (universally) quantified if $\mathcal{Q}_i = \exists$ ($\mathcal{Q}_i = \forall$). The set of existentially (universally) quantified variables is denoted by $var_\exists(F)$ ($var_\forall(F)$). We write $\mathcal{Q}(v)$ to mean $\mathcal{Q}_{\sigma(v)}$. The total ordering $V_1 < \cdots < V_n$ induces a relation of *dominance* $\mathcal{R}^F_{\preceq} \subseteq var(F) \times var(F)$ which is a partial order relation defined as $\langle w, v \rangle \in \mathcal{R}^F_{\preceq}$ iff $w = v$ or $\sigma(w) < \sigma(v)$. When no ambiguity arises, we simply write $w \preceq v$ (read "w dominates v"). The order defined by any $\mathcal{R}^F$ is not total, but *sequentially* total: $v \preceq w$ or $w \preceq v$ whenever $\sigma(v) \neq \sigma(w)$. Given $S \subseteq var(F)$, we define $Sup(S) = \{v \in S | \nexists v' \in S. v \preceq v'\}$. For clauses: $Sup(\Gamma) = \{v \in var(\Gamma) | \nexists v' \in var(\Gamma). v \preceq v'\}$. The universal depth $\delta(v)$ of an existential variable v is the number of dominating universal variables for v: $\delta(v) = |var_\forall(F) \cap \{w | w \preceq v\}|$. We say that $\preceq_2$ is a *restriction* of $\preceq_1$ when $a \preceq_1 b$ implies $a \preceq_2 b$. By fixing an arbitrary order for the variables in each scope, we restrict the relation $\preceq$ and obtain a total ordering. Though the results presented here can be lifted to work with a sequentially total order, we limit to consider the simpler total order case.

3 Building a Quantifier Tree

In this section we characterize a class of QBF instances which lays in between prenex CNF instances and general, unrestricted quantified formulas. In particular, we retain a clause-based negated normal form, but relax the restriction on linearly shaped prefixes. The formulas we consider have a tree-like structure with clauses attached to the leaves.

Our aim is to show that a tree-like formula F^{tree} with certain interesting properties can be extracted efficiently out of any given flat QBF instance F, guaranteeing the key property $F^{tree} \equiv F$. Our starting point is the notion of quantifier tree.

Definition 1 (Quantifier Tree). *A quantifier tree t for a QBF F (whose prefix induces the partial order $\preceq$ among variables) is a labeled tree with the following properties:*

1. *The* root *is labeled by an "and" connective.*
2. *The* internal nodes *are labeled by some variable in F. Existential variables appear once in the tree, while any two internal nodes not laying on the same branch can be labeled by the same universal variable. The labeling of nodes is such that $\preceq$ is a restriction of the relation $\preceq_t$ induced by the tree, where $v \preceq_t w$ iff a descending path from v to w exists in t involving at least one universal and one existential node.*
3. *Each* leaf node *n is labeled with a non-empty set of clauses $G(n) \subseteq F$, and is such that the set of variables encountered along the path from the root to n always contains $var(G(n))$. Every clause in F appears exactly once in the whole tree.*

Let us denote by $trees(F)$ the set of quantifier trees for F. The set $trees(F)$ is never empty: It is easy to verify that at least the tree made up of one single branch linearly replicating the sequence of variables in the prefix of F belongs to $trees(F)$ for every F. In general, further (non-trivial) quantifier trees exist. We focus on such "structured" elements of $trees(F)$. As an example, let us consider the following prenex QBF.

$$\begin{array}{l} \forall a \forall b \exists c \forall d \forall e \exists f \exists g \exists h.(a \vee \neg c) \wedge (a \vee h) \wedge \\ \quad (c \vee \neg d \vee g) \wedge (\neg a \vee b \vee f) \wedge \\ \quad (c \vee e \vee \neg h) \wedge (\neg b \vee \neg f) \wedge (a \vee c \vee \neg g) \end{array} \tag{1}$$

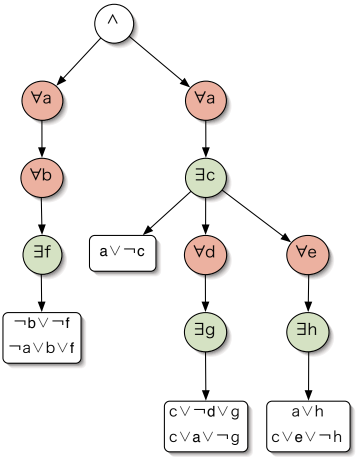


Fig. 1. A quantifier tree for (1)

A quantifier tree for (1) is depicted in Figure 1. It is straightforward to interpret a quantifier tree t as the syntactic tree of a non-prenex quantified boolean formula $qbf(t)$ inductively defined as follows:

$$qbf(n) = \begin{cases} qbf(c_1) \wedge \ldots \wedge qbf(c_n) & \text{for the root} \\ Q_{\sigma(v)} v.\ (qbf(c_1) \wedge \ldots \wedge qbf(c_n)) & \text{for a node labeled by } v \\ \Gamma_1 \wedge \ldots \wedge \Gamma_n & \text{for a leaf labeled by } \{\Gamma_1, \ldots, \Gamma_n\} \end{cases}$$

where $c_1, \ldots, c_n$ are the children of n.

Lemma 1. *For each quantifier tree $t \in trees(F)$, it is $F \equiv qbf(t)$.*

We are interested in extracting quantifiers trees with certain properties. To this end, we employ Algorithm 1. It works in a bottom-up way, growing the tree from the leaves to the root. For the sake of simplicity, we assume that to describe a tree it is sufficient to know the father node $father(n)$ of each node n. Each node n is labeled by a variable $label(n)$ as soon as it is created, and has attached a set of variables $free(n)$ updated at each step to represent the variables that occur free in the current $qbf(n)$. A leaf node n is also labeled by a set of clauses $clauses(n)$. Let us call *miniscoped* a quantifier tree t such that none of the rules in Table 1 can be applied to $qbf(t)$ in a left-to-right direction.

Lemma 2. *Given a prenex QBF F, Algorithm 1 generates a quantifier tree in $trees(F)$.*

Lemma 3. *Algorithm 1 produces miniscoped quantifier trees. It runs in $O(max(|F| \cdot |var(F)|, |var_\exists(F)| \cdot |var(F)|^2))$ requiring $O(|var(F)| \cdot |var_\exists(F)|)$ memory.*

4 How to Profit from a Quantifier Tree

The primary objective of quantifier tree extraction is to help solvers in deciding QBF instances. Most solvers fall into one of three classes, depending on the strategy they follow to attack the problem. We show how each of these approaches benefits from quantifier trees with almost no modification to its core reasoning mechanisms.

Algorithm 1: An algorithm to construct a quantifier tree for a QBF formula

```
input  : A prenex QBF formula $f$
output: A quantifier tree for $f$

// First, we create the root;
$r \leftarrow$ the root node for the tree;
$label(r) \leftarrow$ " $\wedge$ ";

// Then, we create the leaves together with their lists of attached clauses;
$openNodes \leftarrow \emptyset$;
foreach $v \in var_{\exists}(f)$ do
    $n \leftarrow$ new node;
    $label(n) \leftarrow v$;
    $clauses(n) \leftarrow \{\Gamma \in \widetilde{F} | v \in Sup(\Gamma)\}$;
    $free(n) \leftarrow \emptyset$;
    foreach $\Gamma \in clauses(n)$ do
        foreach $\gamma \in \Gamma$ do
            $free(n) \leftarrow free(n) \cup var(\gamma)$;
    $free(n) \leftarrow free(n) \setminus \{v\}$;
    $\widetilde{F} \leftarrow \widetilde{F} \setminus clauses(n)$;
    $openNodes \leftarrow openNodes \cup \{n\}$;

// Finally, the rest of the tree in a bottom-up way;
while $openNodes \neq \emptyset$ do
    $n \leftarrow$ pick one variable from $Sup(openNodes)$;
    if $free(n) = \emptyset$ then
        $father \leftarrow r$;
    else
        $v \leftarrow$ pick one from $Sup(free(n))$;
        if $isUniversal(v)$ then
            $father \leftarrow$ new node;
            $label(father) \leftarrow v$;
            $openNodes \leftarrow openNodes \cup \{father\}$;
            $free(father) \leftarrow free(n) \setminus \{label(n)\}$;
        else
            $father \leftarrow$ the node $n$ with $label(n) = v$ ;
            $free(father) \leftarrow free(father) \cup free(n) \setminus \{label(n)\}$;
    $father(n) \leftarrow father$;
    $openNodes \leftarrow openNodes \setminus \{n\}$;
```

Table 1. Equivalence-preserving FOL rules that move quantifiers

Logical equivalences		Condition
1_a. $\exists x.(\neg\phi) \equiv \neg(\forall x.\phi)$	1_b. $\forall x.(\neg\phi) \equiv \neg(\exists x.\phi)$	
2_a. $\forall x.\phi \equiv \phi$	2_b. $\exists x.\phi \equiv \phi$	$x \notin free(\phi)$
3_a. $\forall x.(\phi \wedge \psi) \equiv (\forall x.\phi) \wedge \psi$	3_b. $\exists x.(\phi \wedge \psi) \equiv (\exists x.\phi) \wedge \psi$	$x \notin free(\psi)$
4_a. $\forall x.(\phi \vee \psi) \equiv (\forall x.\phi) \vee \psi$	4_b. $\exists x.(\phi \vee \psi) \equiv (\exists x.\phi) \vee \psi$	
5_a. $\forall x.(\phi \wedge \psi) \equiv (\forall x.\phi) \wedge (\forall x.\psi)$	5_b. $\exists x.(\phi \vee \psi) \equiv (\exists x.\phi) \vee (\exists x.\psi)$	

4.1 Search-Based Solvers

Search-based solvers extend the DPLL-approach to the quantified case [6]. Examples of solvers in this class are `Quaffle`[14], `QuBE` [8], `ZQSAT` [7], and `semprop` [10]. These algorithms look for models in the most natural way: They follow the left-to-right order of the variables in the prefix during a top-down, depth-first visit of the semantic evaluation tree. Such solvers could be easily lifted to work with the partial order induced by the internal structure of a quantifier tree. The advantage is that nodes with more than one child induce sets of *disjoint* sub-instances to be solved in isolation of one another. For example, suppose we have reached an existential node n with two children, with the assignment Δ along the path from the root to n. The formula to be decided has the following shape: $\exists v.(\phi_1 \wedge \phi_2)$. Once v has got a truth value—say positive—the two instances $\phi_1 * v$ and $\phi_2 * v$ only share universal variables (if any): all the common existential variables have been assigned in Δ, while the clauses in ϕ_1 and ϕ_2 cannot share existential quantifiers by construction. Thus, $\phi_1 * v$ and $\phi_2 * v$ can be solved separately. The resulting situation is depicted for a sample case in Figure 2.

4.2 Resolution-Based Solvers

Rather than search for a model, it is possible to *solve* the formula by applying a *refutationally complete procedure* (`quantor` [4], `QMRES` [12], `QBDD` [12]) that builds upon generalizations of propositional resolution such as *q-resolution* [9, 5]. There are sev-

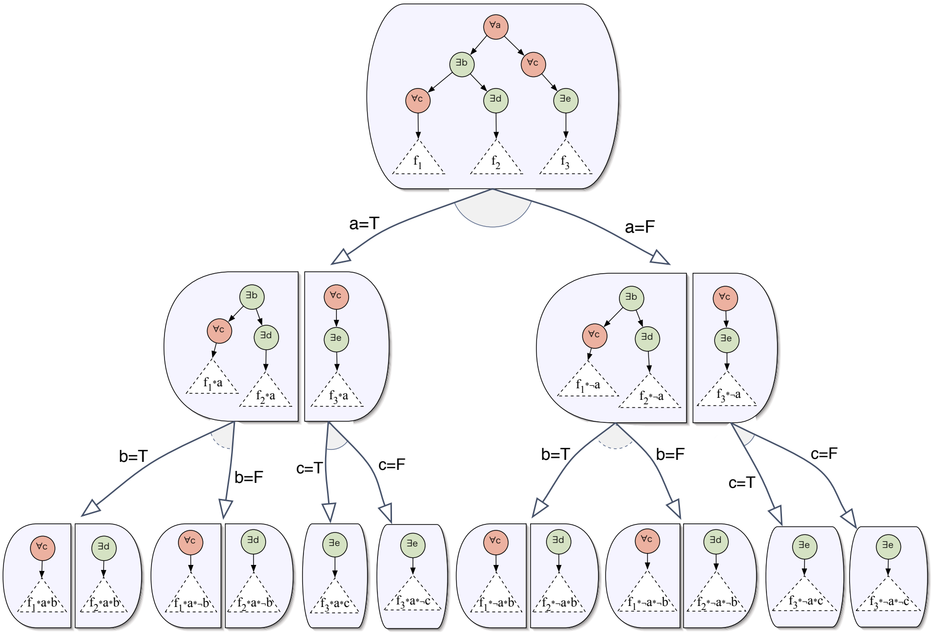


Fig. 2. The top-most part of an AND/OR, divide-et-impera search tree for the quantifier tree $t \in trees(F)$ such that $qbf(t) = \forall a((\exists b(\forall c\, f_1(a, b, c)) \wedge \forall d\, f_2(a, b, d)) \wedge \forall c\forall e\, f_3(a, c, e))$

eral possible complete strategies for applying resolution. For example, we can focus on *eliminating quantifiers* in a right-to-left order (w.r.t. the order in the prefix), getting rid of existential quantifiers by q-resolution. If such solvers were able to exercise brute force against any instance, there would be no point in quantifier trees and, more in general, in heuristic reasoning. Unfortunately, space limits prevent most inference sequences form being feasible, in so as finding a viable solution is the key problem the solver has to face. Quantifier trees help resolution-based solvers in finding better ways to carry on. Let us consider, for example, `quantor` [4]: At each step, it chooses whether to eliminate by q-resolution one of the variables in the deepest existential scope $S_\exists$ or to eliminate by *expansion* one of the variables in the deepest existential scope $S_\forall$. Variables in $S_\forall \cup S_\exists$ are ranked according to a heuristic cost function and the cheapest one is greedily eliminated. With a linear prefix there is only one deepest existential scope and one deepest universal scope. Conversely, within a quantifier tree there are up to one existential/universal scope per branch. This produces two potential benefits: (1) a possibly wider pool of quantifiers to choose from, and (2) one more degree of freedom in the elimination strategy (for example: prefer a quantifier with minimal/maximal depth in the quantifier tree, or focus on one single branch).

4.3 Skolemization-Based Solvers

Skolemization-based solvers replace the original QBF satisfiability problem with the satisfiability problem on the skolemized instance $Sk(F)$. During skolemization, existential quantifiers are eliminated by replacing the variables they bind with *Skolem functions* whose definition domains are appositely chosen to preserve satisfiability. In *outer* skolemization [11] the function introduced for $e \in var_\exists(F)$ depends on all the universal variables that have e in their scope, i.e. (for prenex formulas) all the universal variables to the left of e in the prefix. For example, by outer skolemizing (1) we obtain

$$\begin{array}{l}(a\vee\neg s^c(a,b)) \ \wedge \ (\neg a\vee b\vee s^f(a,b,d,e)) \ \wedge \ (\neg b\vee\neg s^f(a,b,d,e)) \ \wedge \\ \wedge \ (a\vee s^h(a,b,d,e)) \ \wedge \ (s^c(a,b)\vee e\vee\neg s^h(a,b,d,e)) \ \wedge \\ \wedge \ (s^c(a,b)\vee\neg d\vee s^g(a,b,d,e)) \ \wedge \ (a\vee s^c(a,b)\vee\neg s^g(a,b,d,e))\end{array} \tag{2}$$

where s^x is the function with arity $\delta(x)$ introduced to skolemize x, and all the variables are meant to be universally quantified. Once (2) is obtained from (1), methods from FOL automated theorem proving or ad-hoc strategies such as the ones presented in [3] can be employed. Quantifier trees allows us to reduce the arity of the skolem functions. Given that $qbf(t) \equiv F$ for $t \in trees(F)$, we work on $Sk(qbf(t))$ rather than on $Sk(F)$. For example, we obtain for the instance (1):

$$\begin{array}{l}(a\vee\neg s^c(a)) \ \wedge \ (\neg a\vee b\vee s^f(a,b)) \ \wedge \ (\neg b\vee\neg s^f(a,b)) \ \wedge \\ \wedge \ (a\vee s^h(a,e)) \ \wedge \ (s^c(a)\vee e\vee\neg s^h(a,e)) \ \wedge \\ \wedge \ (s^c(a)\vee\neg d\vee s^g(a,d)) \ \wedge \ (a\vee s^c(a)\vee\neg s^g(a,d))\end{array}$$

Simpler skolem functions help the solver in a way that depends on the approach to evaluation it takes. For example, in the case of `sKizzo` [2], each skolemized clause C with m universal variables is translated into (the symbolic representation of) $2^{\delta-m}$ propositional clauses, with $\delta=\delta(C)$ for prenex formulas, and $\delta=\delta_t(C)\leq\delta(C)$ for a quantifier tree t. This produce a linear shrinking of the symbolic representation, and up to an exponential advantage over linear prefixes when ground reasoning is attempted.

5 Experimental Results

We refer to the QBFLIB's archive. Table 2 gives first results on some families of instances. It compares the depth, average universal depth, maximal universal depth, and number of branches before and after tree reconstruction. The time taken to build the tree is also reported. The impact of our reconstruction algorithm over these instances is strong. Conversely, on some other classes (such as *mutex* or *chain*) results are much weaker. The reader may experiment with the downloadable tool `qTree`[2] which takes a QBF as input and produces statistics on the reconstructed tree.

Table 2. The effect of tree-reconstruction over the structure of the syntactic tree

	Before reconstruction				Time	*After reconstruction*			
Instance	*Depth*	*Max &Avg ∀-depth*		*Br.*	(ms)	*Depth*	*Max &Avg ∀-depth*		*Branches*
adder-14-sat	3,641	1,281	1,093.5	1	20	267	94	50.2	28
adder-14-unsat	3,667	665	381.1	1	20	2,988	483	331.8	1
adder-16-sat	4,769	1,672	1,426.4	1	30	307	108	57.4	32
adder-16-unsat	4,799	872	500.6	1	30	3,911	632	434.6	1
Adder2-12-c	11,580	642	603.0	1	60	957	432	414.5	3624
Adder2-12-s	11,580	786	756.9	1	50	116	68	35.3	3624
Adder2-14-c	15,862	875	822.0	1	70	1,296	588	564.2	4956
Adder2-14-s	15,862	1,071	1,031.5	1	70	134	80	41.2	4956
flipflop-7-c	15,213	35	35.0	1	50	1,330	21	20.7	5,121
flipflop-9-c	56,175	45	45.0	1	220	5,466	27	26.9	18,246
flipflop-11-c	159,837	55	55.0	1	720	16,610	33	32.9	50,705
k-branch-n-20	13,822	127	97.9	1	150	5568	127	64.3	2646
k-branch-p-19	12,544	121	93.2	1	130	5063	121	61.3	2400
k-d4-n-16	1,438	69	51.7	1	10	755	69	35.3	310
k-d4-p-19	1,176	62	45.9	1	10	638	62	31.6	250
k-grz-n-18	792	24	17.4	1	10	393	24	11.2	175
k-grz-p-19	767	24	17.7	1	10	379	24	11.5	169
k-ph-n-21	11,131	12	9.7	1	450	5347	12	6.5	1,643
k-ph-p-20	10,444	12	9.7	1	460	5067	12	6.4	1,524
k-poly-n-18	1,465	110	84.0	1	10	926	110	69.1	323
k-poly-p-17	1,384	104	79.4	1	10	875	104	65.4	305
k-t4p-n-19	2,725	123	90.9	1	20	1446	122	61.4	620
k-t4p-p-19	1,470	69	50.6	1	10	782	68	34.5	333
TOILET7.1.iv.13	400	3	2.2	1	10	216	3	1.5	32
TOILET7.1.iv.14	431	3	2.2	1	10	234	3	1.5	32
TOILET10.1.iv.20	855	4	3.0	1	10	457	4	2.0	44
TOILET16.1.iv.32	2,133	4	3.0	1	30	1,117	4	2.0	68
tree-exa10-20	41	20	20.0	1	<1	4	2	2.0	19
tree-exa10-25	51	25	25.0	1	<1	4	2	2.0	24
tree-exa10-30	61	30	30.0	1	<1	4	2	2.0	29

Table 3. Some instances solved on a 900MHz G3 with a 400s timeout

		Solving time (s)	
Instance	*Personality*	*Linear prefix*	*qTree*
tree-exa10-10	B	7.7	0.1
tree-exa10-20	B	timeout	0.2
tree-exa10-30	B	timeout	0.3
TOILET2.1.iv.3	BG	0.3	0.2
TOILET6.1.iv.11	BG	4.0	3.4
TOILET7.1.iv.13	BG	26.4	5.3
k-dup-p-8	QBGS	50.4	0.2
k-dup-p-11	QBGS	54.5	9.7
k-dup-p-15	QBGS	timeout	74.1

		Solving time (s)	
Instance	*Personality*	*Linear prefix*	*qTree*
k-dup-p-16	QBGS	timeout	75.5
k-dup-p-17	QBGS	timeout	327.0
adder-2-sat	B	timeout	0.3
adder-2-sat	BG	timeout	0.9
adder-4-sat	B	timeout	3.0
adder-4-sat	BG	timeout	26.8
adder-6-sat	RS	13.9	10.7
adder-8-sat	RS	57.1	37.4
adder-10-sat	RS	286.5	198.1

Solving advantages depend on the evaluation strategy and on the implementation. We here consider sKizzo [2, 3], a hybrid QBF solver that can be configured to exercise the following strategies (freely combinable to obtain *personalities*): symbolic BDD-based incomplete reasoning (abbreviated in "S"), symbolic resolution-based solving ("R"), branching reasoning with backjumping/learning ("B"), SAT-based solution of ground sub-problems ("G"), and q-resolution reasoning ("Q"). The preliminary results in Table 3 suggest that advantages are expected to cover a broad family of solvers.

6 Conclusions

We presented a method that allows to recover *ex-post* a portion of the internal structure of QBF instances, hidden as a consequence of the cast to prenex normal form. Means to profit from this reconstruction have been presented. Future work on this topic relates to what makes some instances much more sensible to tree reconstruction than others, and to the generalization of our algorithm towards trees with depth minimality properties.

References

1. M. Benedetti. Quantifier Trees for QBFs, Tech.Rep. 05-04-08, ITC-irst, 2005.
2. M. Benedetti. sKizzo's web site, sra.itc.it/people/benedetti/sKizzo, 2004.
3. M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Proc. of LPAR04*, number 3452 in LNCS. 2005.
4. A. Biere. Resolve and Expand. In *Proc. of SAT'04*, pages 238–246, 2004.
5. H. K. Büning and T. Lettmann. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.
6. M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proc. of the tenth conference on Artificial Intelligence/Innovative applications of artificial intelligence*. AAAI, 1998.
7. M. GhasemZadeh, V. Klotz, and C. Meinel. ZQSAT: A QSAT Solver based on Zero-suppressed Binary Decision Diagrams, available at www.informatik.uni-trier.de/TI/bdd-research/zqsat/zqsat.html, 2004.
8. E. Giunchiglia, M. Narizzano, and A. Tacchella. QuBE++: an Efficient QBF Solver. In *Proc. of the 5th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2004.
9. H. Kleine-Buning, M. Karpinski, and A. Flogel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
10. R. Letz. Advances in Decision Procedures for Quantified Boolean Formulas. In *Proceedings of the First International Workshop on Quantified Boolean Formulae (QBF'01)*, pages 55–64, 2001.
11. A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 6, pages 335 – 367. Elsevier, Amsterdam, Netherlands, 2001.
12. G. Pan and M.Y. Vardi. Symbolic Decision Procedures for QBF. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP04)*, 2004.
13. C. Thiffault, F. Bacchus, and T. Walsh. Solving Non-clausal Formulas with DPLL search. In *Proc. of SAT04*, 2004.
14. L. Zhang and S. Malik. Towards Symmetric Treatment of Conflicts And Satisfaction in Quantified Boolean Satisfiability Solver. In *Proc. of CP'02*, 2002.

Quantifier Rewriting and Equivalence Models for Quantified Horn Formulas

Uwe Bubeck[1], Hans Kleine Büning[2], and Xishun Zhao[3]

[1] International Graduate School Dynamic Intelligent Systems,
Universität Paderborn, 33098 Paderborn, Germany
bubeck@upb.de

[2] Department of Computer Science, Universität Paderborn,
33098 Paderborn, Germany
kbcsl@upb.de

[3] Institute of Logic and Cognition, Zhongshan University,
510275, Guangzhou, P.R. China
hsdp08@zsu.edu.cn

Abstract. In this paper, quantified Horn formulas with free variables ($QHORN^*$) are investigated. The main result is that any quantified Horn formula Φ of length $|\Phi|$ with free variables, $|\forall|$ universal quantifiers and an arbitrary number of existential quantifiers can be transformed into an equivalent formula of length $O(|\forall| \cdot |\Phi|)$ which contains only existential quantifiers. Moreover, it is shown that quantified Horn formulas with free variables have equivalence models where every existential quantifier is associated with a monotone Boolean function.

The results allow a simple representation of quantified Horn formulas as purely existentially quantified Horn formulas ($\exists HORN^*$). An application described in the paper is to solve $QHORN^*$-SAT in $O(|\forall| \cdot |\Phi|)$ by using this transformation in combination with a linear-time satisfiability checker for propositional Horn formulas.

1 Introduction

Quantified Boolean Formulas (QBF) offer a concise way to represent formulas which arise in areas such as planning, scheduling or verification. The ability to provide compact representations for many Boolean functions does however come at a price: determining the satisfiability of formulas in QBF is PSPACE-complete, which is assumed to be significantly harder than the NP-completeness of the propositional SAT problem. However, continued research and the lifting of propositional SAT techniques to $QBFs$ have recently produced interesting improvements (see, e.g., [2, 3, 9, 10]) and have lead to the emergence of more powerful QBF-SAT solvers [8].

Furthermore, the satisfiability problem is known to be tractable for some restricted subclasses like $QHORN$ [5] or $Q2-CNF$ [1]. Those classes are defined by imposing restrictions on the syntactic structure of the formula matrices. The

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 386–392, 2005.

interesting question which we are investigating in this paper is how such a syntactic restriction is affecting the structure of the set of satisfying truth value assignments to the existentially quantified variables.

A suitable concept for describing the satisfiable truth value assignments to the existential variables is the notion of models for formulas in QBF, which has been introduced in [6]: for a quantified formula Φ with existential variables $\mathbf{y} = y_1, ..., y_m$, let $M = (f_{y_1}, ..., f_{y_m})$ be a mapping which maps each existential variable y_i to a propositional formula f_{y_i} over universal variables whose quantifiers precede the quantifier of y_i. Then M is a satisfiability model for Φ if the resulting formula $\Phi[\mathbf{y}/M] := \Phi[y_1/f_{y_1}, ..., y_m/f_{y_m}]$, where simultaneously each existential variable y_i is replaced by its corresponding formula f_{y_i} and the existential quantifiers are dropped from the prefix, is true.

The concept of models for closed formulas can easily be extended to quantified Boolean formulas with free variables (QBF^*) [7]. In that case, the propositional formulas f_{y_i} may also contain free variables. For QBF^* formulas, an interesting additional constraint is to require that Φ and $\Phi[\mathbf{y}/M]$ must be equivalent. Formally, *equivalence models* are defined as follows: let $\Phi(\mathbf{z}) = Q\,\phi(\mathbf{x}, \mathbf{y}, \mathbf{z})$ be a quantified Boolean formula with prefix Q and matrix ϕ, universal variables $\mathbf{x} = x_1, ..., x_n$, existential variables $\mathbf{y} = y_1, ..., y_m$ and free variables $\mathbf{z} = z_1, ..., z_r$. For propositional formulas f_{y_i} over $\mathbf{z}$ and over universal variables whose quantifiers precede $\exists y_i$, we say $M = (f_{y_1}, ..., f_{y_m})$ is an equivalence model for $\Phi(\mathbf{z})$ if and only if $\Phi(\mathbf{z}) \approx \forall x_1 ... \forall x_n\, \phi(x_1, ..., x_n, \mathbf{y}, \mathbf{z})[\mathbf{y}/M]$.

In this paper, we focus on the class of quantified Horn formulas with free variables ($QHORN^*$). While it has been previously known that closed quantified Horn formulas have equivalence models consisting of monotone monomials and the constant functions 0 or 1 (see [6]), the structure of $QHORN^*$ equivalence models has been an open problem. In Section 2, we show that $QHORN^*$ formulas have equivalence models where the propositional formulas f_{y_i} contain only conjunctions and disjunctions of positive literals, i.e. the f_{y_i} are monotone.

In the second part of this paper, we turn to the universal quantifiers. Section 3 demonstrates that we can eliminate all universal quantifiers and the corresponding universal variables in $QHORN^*$ formulas without significantly increasing the length of the formula. To be more precise, we show that a quantified Horn formula Φ of length $|\Phi|$ with free variables, $|\forall|$ universal quantifiers and an arbitrary number of existential quantifiers can be transformed into an equivalent formula of length $O(|\forall| \cdot |\Phi|)$ which contains only existential quantifiers.

We finally explain how this transformation can be used to solve the satisfiability problem for $QHORN^*$ in time $O(|\forall| \cdot |\Phi|)$ with a very simple algorithm.

We need some additional notation: for $\Phi \in QBF^*$, $\Phi(\mathbf{z}) = Q\,\phi(\mathbf{x}, \mathbf{y}, \mathbf{z})$ with $Q = \forall x_{1,1} ... \forall x_{1,n_1} \exists y_{1,1} ... \exists y_{1,m_1} ... \forall x_{r,1} ... \forall x_{r,n_r} \exists y_{r,1} ... \exists y_{r,m_r}$, $n_i \geq 1$ and $m_i \geq 1$ for $i = 1, .., r$, we combine successive quantifiers of the same kind and simply write $Q = \forall X_1 \exists Y_1 ... \forall X_r \exists Y_r$ with $X_i = (x_{i,1}, ..., x_{i,n_i})$ and $Y_i = (y_{i,1}, ..., y_{i,m_i})$. Another notation that we use is $ab := (a_1, ..., a_m, b_1, ..., b_n)$ to denote the concatenation of two tuples $a = (a_1, ..., a_m)$ and $b = (b_1, ..., b_n)$.

2 Equivalence Models for $QHORN^*$ Formulas

Definition 1. *Let $M = (f_{y_1}, ..., f_{y_m})$ be an equivalence model for a quantified Boolean formula $\Phi \in QBF^*$. Then M is a* ***monotone equivalence model*** *if and only if the functions f_{y_i}, $1 \leq i \leq m$, do not contain negative occurrences of atoms, i.e. the f_{y_i} can be written using only positive literals and the reduced operator set $\{\wedge, \vee\}$ as well as $f_{y_i} = 0$ or $f_{y_i} = 1$.*

Theorem 1. *Any formula $\Phi \in QHORN^*$ has a monotone equivalence model $M = (f_{y_1}, ..., f_{y_m})$ which satisfies the following properties: $\Phi[\mathbf{y}/M] \in QHORN^*$, and any clause in Φ with a positive existential variable contributes only tautological clauses to $\Phi[\mathbf{y}/M]$.*

Proof: Due to space limitations, we omit the proof. It is by induction on the number of quantifiers and can be outlined as follows: in the induction base, when there is only one single existential variable, a monotone model is chosen such that tautological clauses are created when the model is substituted for positive instances of the existential variable. On the other hand, the negative instances produce the set of possible Q-resolvents, which is known to be equivalent to the original formula. In the induction step, we compose a monotone equivalence model for $k-1$ existential variables and a monotone equivalence model for the k-th existential variable to obtain a model for all k existential variables.

Unfortunately, these models may be of exponential length, as a short argument reveals. It is known (see, e.g., [5]) that converting a quantified Horn formula into an equivalent propositional Horn formula may result in an exponential increase in length required. But if we had an equivalence model of polynomial length, we could use it to eliminate the existential variables, which in turn would allow us to drop the universal variables, and we would end up with a purely propositional formula of polynomial length. That contradiction leads to the conclusion that the equivalence model may have exponential length. Nevertheless, knowing about the monotony of those models is useful, as it allows for concise and elegant proofs.

3 Eliminating Universal Quantifiers

As converting a quantified Horn formula into an equivalent propositional Horn formula may result in an exponentially longer formula, it is not a practical way to go. As discussed above, eliminating only the existential quantifiers is not practical either. But as it turns out, eliminating just the universal quantifiers does not lead to this exponential increase in length. We will now prove this surprising result and present the corresponding algorithm.

Definition 2. *A formula $\Phi \in QHORN^*$ is an* ***existentially quantified Horn formula*** *with free variables if it is of the form $\Phi(\mathbf{z}) = \exists y_1 ... \exists y_m \, \phi(\mathbf{z})$ $(m \geq 0)$, i.e. if it does not contain universally quantified variables. The class of all such formulas we denote by $\exists HORN^*$.*

The goal of the following investigation is to transform an arbitrary formula in $QHORN^*$ into an equivalent formula in $\exists HORN^*$ with a polynomial increase in length. The method that we present is a specialization of the known exponential method of expanding universal quantifiers in general QBF^* formulas. We first present the general technique and then show how it can be refined for $QHORN^*$.

3.1 Eliminating Universal Quantifiers in QBF^* Formulas

The general method for expanding universal quantifiers is rather straightforward: two copies of the original matrix are generated, one for the universally quantified variable being true, and one for that variable being false. As explained in [2], existential variables which depend on that universal variable need to be duplicated as well. For example, the formula $\exists y_1 \forall x \exists y_2\, \phi(x, y_1, y_2)$ is expanded to $\exists y_1 \exists y_2 \exists y_2'\, \phi(0, y_1, y_2) \wedge \phi(1, y_1, y_2')$. For multiple universal quantifiers, we successively expand each universal quantifier, starting with the innermost.

Based on this informal description, we now provide a formal representation. Let $\Phi \in QBF^*$ with $\Phi(\mathbf{z}) = \forall X_1 \exists Y_1 ... \forall X_r \exists Y_r\, \phi(X_1, ..., X_r, Y_1, ..., Y_r, \mathbf{z})$, where $X_i = (x_{i,1}, ..., x_{i,n_i})$ and $Y_i = (y_{i,1}, ..., y_{i,m_i})$ ($n_i \geq 1$ and $m_i \geq 1$, $i = 1, ..., r$, $r \geq 1$). Without loss of generality, we assume that the outermost quantifiers are universal and the innermost quantifiers are existential.

The expanded formula is then given as$\Phi_{\exists \exp}(\mathbf{z}) :=$

$$\bigwedge_{A_1 \in \{0,1\}^{n_1}} \exists Y_1^{A_1} \left(\bigwedge_{A_2 \in \{0,1\}^{n_2}} \exists Y_2^{A_2} .... \left(\bigwedge_{A_r \in \{0,1\}^{n_r}} \exists Y_r^{A_r}\, \phi(A_1 ... A_r, Y_1 ... Y_r, \mathbf{z}) \right) ... \right)$$

The tuples A_i represent the possible truth value assignments to the universal variables $x_{i,1}, ..., x_{i,n_i}$. The expression $\bigwedge_{A_i \in \{0,1\}^{n_i}}$ should be understood as a conjunction of 2^{n_i} clauses, one for each truth value assignment. Finally, $\exists Y_i^{A_i}$ is an abbreviation for $\exists y_{i,1}^{A_i} ... \exists y_{i,m_i}^{A_i}$, the copies of the i-th sequence of existential quantifiers. The index A_i is used to have a unique name for each of those copies. For example, four copies of $y_{i,j}$ would be named $y_{i,j}^{(0,0)}$, $y_{i,j}^{(0,1)}$, $y_{i,j}^{(1,0)}$ and $y_{i,j}^{(1,1)}$.

If there are n universal quantifiers in a formula Φ, its expansion $\Phi_{\exists \exp}$ contains 2^n copies of the formula's original matrix. This exponential increase in length makes the method unusable in general, but we can significantly simplify it:

3.2 Special Case: $QHORN^*$ Formulas

Definition 3. *Let* $\Phi(\mathbf{z}) = \forall X_1 \exists Y_1 ... \forall X_r \exists Y_r\, \phi(X_1, ..., X_r, Y_1, ..., Y_r, \mathbf{z})$ *with* $X_i = (x_{i,1}, ..., x_{i,n_i})$ *and* $Y_i = (y_{i,1}, ..., y_{i,m_i})$ *(*$n_i \geq 1$ *and* $m_i \geq 1$*,* $i = 1, ..., r$*,* $r \geq 1$*) be a quantified Horn formula whose outermost quantifiers are universal and whose innermost quantifiers are existential.*

Then we define the formula $\Phi_{\exists\mathrm{poly}}(\mathbf{z})$ as

$$\Phi_{\exists\mathrm{poly}}(\mathbf{z}) := \bigwedge_{A_1 \in Assign_1} \exists Y_1^{A_1}$$

$$\left(\bigwedge_{A_2 \in Assign_2(A_1)} \exists Y_2^{A_2} \ldots. \left(\bigwedge_{A_r \in Assign_r(A_1 \ldots A_{r-1})} \exists Y_r^{A_r}\, \phi(A_1 \ldots A_r, Y_1 \ldots Y_r, \mathbf{z}) \right) \ldots \right)$$

with the restricted set of possible assignments

$$Assign_1 = MaxOneZero(n_1)$$

$$Assign_i(A_1, \ldots, A_{i-1}) = \begin{cases} MaxOneZero(n_i) & \text{, if } A_1 \ldots A_{i-1} = \{1\}^{n_1 + \ldots + n_{i-1}} \\ \{1\}^{n_i} & \text{, else} \end{cases}$$

where

$$MaxOneZero(n) = \{(a_1, \ldots, a_n) \mid \exists i : \ a_i = 0 \,\text{and}\, a_j = 1 \,\text{for}\, j \neq i\} \cup \{(1, \ldots, 1)\}$$

is the set of n-ary tuples of binary values with at most one component being zero.

The only difference between the formula $\Phi_{\exists\mathrm{poly}}$ and the expansion $\Phi_{\exists\mathrm{exp}}$ for general QBF^* formulas is that for quantified Horn formulas, not all possible truth value assignments to the universally quantified variables have to be considered. For Horn formulas, we discard assignments where more than one universally quantified variable is false.

Lemma 1. *$\Phi_{\exists\mathrm{poly}}$ is equivalent to Φ.*

Proof: We need to show that for a quantified Horn formula Φ with free variables $\mathbf{z}$, $\Phi_{\exists\mathrm{exp}}(\mathbf{z}) \approx \Phi_{\exists\mathrm{poly}}(\mathbf{z})$ holds. Due to space considerations, we also have to omit this proof. The main proof idea is as follows: if the matrix of $\Phi_{\exists\mathrm{poly}}$ can be satisfied when exactly one universal variable is false, we can compensate for an additional universal variable being false by modifying the truth value assignment to the existential variables.

In the definition of $\Phi_{\exists\mathrm{poly}}$, we can observe that there is one instantiation of the matrix of the original formula for each possible assignment to the universal variables where either all of those variables are true, or exactly one of them is false. There are $n + 1$ such assignments. Furthermore, the previous lemma has shown that $\Phi_{\exists\mathrm{poly}}$ is equivalent to $\Phi_{\exists\mathrm{exp}}$, which in turn is equivalent to Φ, so we immediately have the following theorem:

Theorem 2. *For any quantified Horn formula $\Phi \in QHORN^*$ with free variables, there exists an equivalent formula $\Phi' \in \exists HORN^*$ without universal quantifiers. The length of Φ' is bounded by $|\forall| \cdot |\Phi|$, where $|\forall|$ is the number of universal quantifiers in Φ, and $|\Phi|$ is the length of Φ.*

3.3 The Transformation Algorithm

Listing 1 presents an algorithm to transform Φ into $\Phi_{\exists\mathrm{poly}}$ as described above. In the main loop of the algorithm, one universal variable $x_{i,j}$ is given the value false, while all the others are true. For any such assignment $A_{\mathbf{x}}$, all universal variables which are dominated by $x_{i,j}$ (i.e. their corresponding quantifier follows $\forall x_{i,j}$) have to be replaced by independent new variables $\mathbf{y}'$. Then, the matrix of the original formula has to be duplicated, with $A_{\mathbf{x}}$ being substituted for $\mathbf{x}$ and $\mathbf{y}'$ being substituted for $\mathbf{y}$. After executing the main loop, one additional copy is needed for the case where all universal variables are true.

Listing 1: The Transformation Algorithm

```
// Input: $\Phi \in QHORN^*$, $\Phi(\mathbf{z}) = \forall X_1 \exists Y_1 ... \forall X_r \exists Y_r\, \phi(X_1, ..., X_r, Y_1, ..., Y_r, \mathbf{z})$,
//        where $X_i = (x_{i,1}, ..., x_{i,n_i})$ and $Y_i = (y_{i,1}, ..., y_{i,m_i})$
// Output: $\Phi_{\exists poly} \in \exists HORN^*$ with $\Phi_{\exists poly} \approx \Phi$

$\phi_{\exists poly}$ = $\emptyset$;
for ($i$ = 1 to $r$) do
  for ($j$ = 1 to $n_i$) do $A_{x_{i,j}}$ = 1;
for ($i$ = 1 to $r$) do {
  for ($j$ = 1 to $n_i$) do {
    $A_{x_{i,j}}$ = 0;
    for ($k$ = $i$ to $r$) do
      for ($l$ = 1 to $m_k$) do $y'_{k,l}$ = new $\exists$-var;
    $\phi_{\exists poly}$ = $\phi_{\exists poly} \cup \phi[\mathbf{x}/A_{\mathbf{x}}, \mathbf{y}/\mathbf{y}']$; // (*)
    $A_{x_{i,j}}$ = 1;
  }
  for ($l$ = 1 to $m_i$) do $y'_{i,l}$ = new $\exists$-var;
}
$\phi_{\exists poly}$ = $\phi_{\exists poly} \cup \phi[\mathbf{x}/A_{\mathbf{x}}, \mathbf{y}/\mathbf{y}']$; // (*)
```

The lines marked with (*) need time $O(|\Phi|)$. They are executed $n_1 + ... + n_r + 1 = |\forall| + 1$ times, so the algorithm in total requires time $O(|\forall| \cdot |\Phi|)$.

3.4 Application: Satisfiability Testing

Let $\Phi(\mathbf{z}) \in \exists HORN^*$ be an existentially quantified Horn formula of the form $\Phi(\mathbf{z}) = \exists y_1 ... \exists y_m\, \phi(y_1, ..., y_m, \mathbf{z})$. Then $\Phi(\mathbf{z})$ is satisfiable if and only if its matrix $\phi(y_1, ..., y_m, \mathbf{z})$ is satisfiable. The latter is a purely propositional formula, therefore a SAT solver for propositional Horn formulas can be used to determine the satisfiability of an arbitrary formula in $\exists HORN^*$.

That observation leads to the following algorithm for determining the satisfiability of a formula $\Psi \in QHORN^*$:

1. Transform Ψ into $\Psi_{\exists\mathrm{poly}} \in \exists HORN^*$ with $|\Psi_{\exists\mathrm{poly}}| = O(|\forall| \cdot |\Psi|)$. This requires time $O(|\forall| \cdot |\Psi|)$ as discussed in Section 3.3.

2. Determine the satisfiability of $\psi_{\exists\text{poly}}$, which is the purely propositional matrix of $\Psi_{\exists\text{poly}}$. It is well known (see [4]) that SAT for propositional Horn formulas can be solved in linear time, in this case $O(|\psi_{\exists\text{poly}}|) = O(|\forall| \cdot |\Psi|)$.

In total, the algorithm requires time $O(|\forall| \cdot |\Psi|)$. The best existing algorithm presented in [5] has the same complexity, but is significantly more complicated.

4 Conclusion and Outlook

In this paper, we have given two new properties of $QHORN^*$ formulas:

(1) they can easily be transformed into equivalent purely existentially quantified formulas of length $O(|\forall| \cdot |\Phi|)$, (2) they have monotone equivalence models. Both properties characterize $QHORN^*$ as a rather simple subclass of QBF^* formulas.

Based on these results, a new algorithm for determining the satisfiability of $QHORN^*$ formulas in time $O(|\forall| \cdot |\Phi|)$ has been presented. Further investigation is needed to determine whether the simplicity of this new algorithm also translates to better real-world performance than the previously known algorithm.

References

[1] B. Aspvall, M. Plass, and R. Tarjan. *A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas.* Information Processing Letters, 8(3):121–123, 1979.

[2] A. Biere. *Resolve and Expand.* Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04). Springer LNCS, 2004.

[3] M. Cadoli, A. Giovanardi, and M. Schaerf. *An algorithm to evaluate quantified boolean formulae.* Proc. 16th Natl. Conf. on AI (AAAI-98), 1998.

[4] W. Dowling and J. Gallier. *Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae.* J. of Logic Programming, 1(3):267–284, 1984.

[5] H. Kleine Büning and T. Lettmann. *Propositional Logic: Deduction and Algorithms.* Cambridge University Press, Cambridge, UK, 1999.

[6] H. Kleine Büning, K. Subramani, and X. Zhao. *On Boolean Models for Quantified Boolean Horn Formulas.* Proc. 6th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'03). Springer LNCS 2919 93–104, 2004.

[7] H. Kleine Büning and X. Zhao. *Equivalence Models for Quantified Boolean Horn Formulas.* Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04). Springer LNCS, 2004.

[8] D. Le Berre, M. Narizzano, L. Simon, and A. Tacchella. *The second QBF solvers comparative evaluation.* Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04). Springer LNCS, 2004.

[9] R. Letz. *Advances in Decision Procedures for Quantified Boolean Formulas.* Proc. IJCAR 2001 Workshop on Theory and Applications of QBF, 2001.

[10] J. Rintanen. *Partial implicit unfolding in the Davis-Putnam procedure for quantified boolean formulae.* Intl. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'01), 2001.

A Branching Heuristics for Quantified Renamable Horn Formulas

Sylvie Coste-Marquis, Daniel Le Berre, and Florian Letombe[*]

CRIL, CNRS FRE 2499
{coste, leberre, letombe}@cril.univ-artois.fr

Abstract. Many solvers have been designed for $\mathcal{QBF}$s, the validity problem for Quantified Boolean Formulas for the past few years. In this paper, we describe a new branching heuristics whose purpose is to promote renamable Horn formulas. This heuristics is based on Hébrard's algorithm for the recognition of such formulas. We present some experimental results obtained by our QBF solver **Qbfl** with the new branching heuristics and show how its performances are improved.

1 Introduction

QBF, the validity problem for $\mathcal{QBF}$s, has a growing importance in AI. This can be explained by the fact that, as the canonical PSPACE-complete problem, many AI problems can be polynomially reduced to QBF; furthermore, there is some empirical evidence from various AI fields (including among others planning, nonmonotonic reasoning, paraconsistent inference) that a translation-based approach can prove more "efficient" than domain-dependent algorithms dedicated to such AI tasks (see [1, 2, 3, 4, 5]). Accordingly, many QBF solvers have been designed and evaluated for the past few years (see mainly [6, 7, 8, 9, 10, 11, 12, 13, 14]).

Among the few approaches to increase the set of instances that are feasible from the practical point of view are the *restriction-based* ones. The key idea is to recognize instances for which specific algorithms can prove much more efficient than general QBF solvers. Several tractable restrictions of $\mathcal{QBF}$ have been identified so far. [15, 16] show that quantified Krom formulas ($\mathcal{QKF}$) are polynomially solvable. [17] proved that quantified Horn formulas ($\mathcal{QHF}$) form a tractable restriction of $\mathcal{QBF}$. As an easy consequence, quantified renamable Horn formulas (ren$\mathcal{QHF}$s for short), are polynomially solvable too.

The main objective of this paper is to present a new branching heuristics for QBF solvers based on the DPLL procedure. This new heuristics aims at promoting the generation of quantified renamable Horn formulas, for which efficient solvers exist. The rest of this paper is organized as follows. Some formal preliminaries are given in Section 2. The new heuristics is presented in Section 3. Experimental results obtained by our QBF solver with this heuristics are reported in section 4. Finally, Section 5 concludes the paper and gives some perspectives.

[*] Thanks to anonymous reviewers for their remarks. This work has been supported in part by UIT de Lens and Université d'Artois.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 393–399, 2005.

2 Quantified Boolean Formulas

Let $Var(\Sigma)$ be the set of variables of the propositional formula Σ. A $\mathcal{QBF}$ formula Σ is said to be *prenex* if and only if $\Sigma = Qx_1(\ldots Qx_n(\phi)\ldots)$ where each occurrence of Q stands for either $\forall$ or $\exists$, and ϕ does not contain any quantified occurrence of a variable. ϕ is said to be the *matrix* of Σ and the sequence $Qx_1 \ldots Qx_n$ of quantifications is the *prefix* of Σ. A formula is said to be *closed* if and only if it has no free variable. In this paper, we always consider a $\mathcal{QBF}$ in prenex form with a $\mathcal{CNF}$ matrix. We focus in this paper on two restrictions of $\mathcal{QBF}$s: $\mathcal{QHF}$s and ren$\mathcal{QHF}$s.

Definition 1 (Quantified Horn Formula). *Clauses of a $\mathcal{QHF}$ contain at most one positive literal.*

Our branching heuristics oriented towards the generation of ren$\mathcal{QHF}$s is based on Hébrard's recognition of renamable Horn formulas [18]. Before presenting it, we first need to recall some notions introduced by Hébrard. Let Σ be a $\mathcal{QBF}$ and let L be a set of literals. L is *consistent* if it does not contain p and $\neg p$ for every propositional variable p. L is *complete* if p belongs to L or $\neg p$ belongs to L, for all p in $Var(\Sigma)$. A *renaming R* is a complete and consistent set of literals. R is a Horn renaming for Σ if a $\mathcal{QHF}$ formula is obtained when replacing in the matrix of Σ every occurrence of a literal by its complementary literal if the negative literal belongs to R.

Definition 2 ($\Rightarrow$, $\Rightarrow^*$ and $\textsc{Clos}(l)$).
Let l and t be two literals and Σ a $\mathcal{CNF}$ formula being the matrix of a $\mathcal{QBF}$.

- *$l \Rightarrow t$ iff $\exists$ a clause $C \in \Sigma$ s.t. $l \in C$, $\neg t \in C$ and $l \neq \neg t$.*
- *The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.*
- *$\textsc{Clos}(l)$ denotes the set $\{t \mid l \Rightarrow^* t\}$.*

A set of literals L is said to be *closed* if $\textsc{Clos}(l) \subseteq L$, $\forall l \in L$.

Proposition 1. *A renaming R is a Horn renaming if and only if it is closed, i.e. $\forall l \in R$, $\textsc{Clos}(l) \subseteq R$(Proposition 1.1 from [18]).*

A sketch of Hébrard's variables renaming algorithm is presented in algorithm 1.1.

3 A New Branching Heuristics

The idea of our heuristics is to branch first on variables whose propagation leads to a renamable Horn formula, or a formula that is "almost" renamable Horn. We use Hébrard's algorithm to detect renamable Horn formulas. This algorithm computes a set of literals to be renamed in order to obtain a Horn formula. If such a set exists, then we can use Kleïne-Büning polynomial time algorithm [17] in order to solve the instance. Otherwise, we use the algorithm to measure how far we are from a ren$\mathcal{QHF}$: the greater the measure is, the closer we are from a ren$\mathcal{QHF}$. We call that measure the *Contradiction's distance.*

Algorithme 1.1: Hébrard's Horn renaming algorithm (on the left) and measure of the contradiction's distance δ (on the right)

```
function RHClos
Input : a $\mathcal{QBF}$ $\mathbf{Q}$
Output : $\emptyset$ if $\mathbf{Q}$ is not Horn renamable,
         else a Horn renaming $R$ for $\mathbf{Q}$.
begin
    $R \leftarrow \emptyset$ ;
    foreach variable p in Var(Q) do
        if $p \notin R$ and $\neg p \notin R$ then
            if $\forall q \in \textsc{Clos}(p)\backslash R, \neg q \notin \textsc{Clos}(p)\backslash R$ then
                # $\textsc{Clos}(p)\backslash R$ is consistent
                $R \leftarrow R \cup (\textsc{Clos}(p)\backslash R)$ ;
            else if $\forall q \in \textsc{Clos}(\neg p)\backslash R, \neg q \notin \textsc{Clos}(\neg p)\backslash R$ then
                # $\textsc{Clos}(\neg p)\backslash R$ is consistent
                $R \leftarrow R \cup (\textsc{Clos}(\neg p)\backslash R)$ ;
            else return $\emptyset$ ;
end
```

```
function $\delta$
Input : a $\mathcal{QBF}$ $\mathbf{Q}$, a litral $l$
Output : the distance $\delta$ of $l$ to a Horn renaming.
begin
    $Closl \leftarrow \{l\}$ ;
    $Distance \leftarrow 0$ ;
    $\textsc{Put}(lQueue, l)$ ;
    while not $\textsc{EmptyQueue}(lQueue)$ do
        $m \leftarrow \textsc{Get}(lQueue)$ ;
        $LeadsTo \leftarrow \{\neg t | m \Rightarrow t\}$ ;
        foreach $e \in LeadsTo$ do
            if $\neg e \in Closl$ then
                return $Distance + 1$ ;
            else if $e \in Closl$ then
                $Closl \leftarrow Closl \cup \{e\}$ ;
                $\textsc{Put}(lQueue, e)$ ;
        $Distance \leftarrow Distance + 1$ ;
    return $Distance + 1$ ;
end
```

Definition 3 (Contradiction's distance). *The distance from a contradiction of Horn renamability δ_l for a literal l in a $\mathcal{QBF}$ Q is defined as follows:*

$$\delta_l = \begin{cases} 0 \; if \; \nexists v | l \Rightarrow v \\ 1 \; if \; l \Rightarrow t \; and \; l \Rightarrow \neg t \\ 1 + min(\delta_v | l \Rightarrow v) \; otherwise. \end{cases}$$

The idea behind that distance is that the closer we are to a renamable Horn formula, the greater the contradiction distance should be.

In order to compute that distance, we need to slightly modify Hébrard's algorithm: Hébrard's algorithm performs DFS (Depth-First Search) while ours use BFS (Breadth-First Search). This is mandatory to ensure that the value returned by the algorithm is minimal. The right part of the algorithm 1.1 describes such a computation. It's in linear complexity ($O(n+m)$, with n the number of literals and m the number of clauses, considering that size of the clauses is bounded by a constant).

This distance is often not sufficient to elect a single variable. As a consequence, we refine it using the well-known two-sided Jeroslow-Wang heuristics [19] with a setting widely used in complete solvers for random k-SAT inherited from POSIT [20]. We restrict the selection of the variables to the outermost quantifier scope and we branch first on the literal having the greatest contradiction distance.

Definition 4 (Heuristics Δ of Horn Renamability). *Let X_1 be the outermost quantifier group. Our heuristic function Δ is as follows:*

- $\Delta_x = 1024 \times \delta_x \times \delta_{\neg x} + \delta_x + \delta_{\neg x}$ *;*
- *the variable x is chosen if $x \in X_1$ and, $\forall y \in X_1$, $(x \neq y \Rightarrow \Delta_x \leq \Delta_y)$;*
- *the literal x is finally chosen if $\delta_x > \delta_{\neg x}$, $\neg x$ otherwise.*

4 Experimental Results

4.1 Methodology

The empirical results presented in this section have been obtained on PIV 3GHz computers with 512MB of RAM. In order to compare Qbfl (version 1.7) with other solvers, we chose to use QuBE-Rel[1] (version 1.3) and Semprop[2] (version 010604) for those experiments since those two solvers are two state-of-the-art QBF solvers according to the recent $\mathcal{QBF}$ evaluations[3] [21] that are freely available. In the following tables, only the CPU time of instances solved is taken into account. As a result, it is necessary to also check the number of benchmarks solved in order to compare the behavior of the solvers.

4.2 "Polynomial" Benchmarks

Our first experiments are on sets of randomly generated $\mathcal{QHF}$ and ren$\mathcal{QHF}$ benchmarks[4] The instances are generated as follows:

$\mathcal{QHF}$ for each clause, we randomly choose the size of the clause. Then, a literal is chosen to be the positive one in the clause. We complete it by randomly choosing negative literals;

ren$\mathcal{QHF}$ we first chose the number of variables to be renamed then those variables are chosen before generating clauses using the same method as the one used for generating $\mathcal{QHF}$s but renaming the chosen literals.

We chose to use only two alternations of quantifiers, $\forall X \exists Y$, such that $|X| > \alpha$ with $2 < \alpha < 2/3 \times \#V$ with $\#V$ the number of variables of the formula.

Those benchmarks are sorted by groups of 1000 (resp. 100) instances of $\mathcal{QHF}$s (resp. ren$\mathcal{QHF}$s). Each instance has 400 variables and we increase the usual clauses over variables ratio from 3 to 6. The timeout is fixed at 60 seconds. Note that this timeout is 3 orders of magnitude greater than the CPU time needed by our dedicated solver for those benchmarks and that similar results were obtained using a timeout of 300s.

Figure 1 reports empirical comparisons on ren$\mathcal{QHF}$s. The behavior of the solvers is quite similar to the $\mathcal{QHF}$case. However, those benchmarks look more difficult for **Semprop** which fails on 10% of them.

Note that **Qbfl** does not appear on the graph on the right hand side because our solver solved all those benchmarks.

We can observe that the runtime used by **Qbfl** to solve those polynomial formulas increases slowly and regularly. Those first experiments do not show that our approach is useful: They only show that our solver can detect and solve $\mathcal{QHF}$ and ren$\mathcal{QHF}$ formulas quite efficiently and that those formulas are not trivial for current state-of-the-art QBF solvers. We submitted to the 2005 $\mathcal{QBF}$

[1] http://www.star.dist.unige.it/~qube.

[2] http://www4.in.tum.de/~letz/semprop.

[3] Some results of the evaluations are available on http://www.qbflib.org/qbfeval.

[4] Generators are available at http://www.cril.uni-artois.fr/~letombe/qbfg.

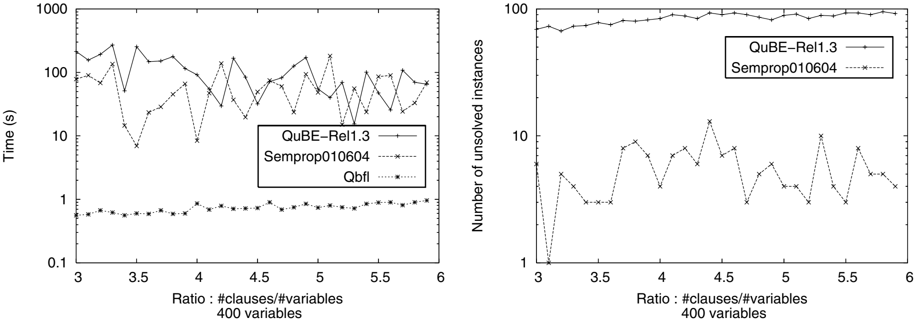


Fig. 1. Comparison between Semprop, QuBE and Qbfl on sets of 100 renamable Horn random instances

Evaluation the benchmarks on which **Semprop** failed. It is more interesting to see how our solver behave on last year $\mathcal{QBF}$ evaluation set of benchmarks.

4.3 QBF 2004 Evaluation Benchmarks

We focus first on formulas proposed by G. Pan, translated from modal logic K from TANCS'98 [22]. Those 378 instances - 9 types of 21 benchmarks valid and 9 not valid - have been proposed for 2003 $\mathcal{QBF}$ solvers evaluation and most of them were classified as *hard*[5]. Since 2003, solvers have improved and in the 2004 $\mathcal{QBF}$ evaluation classification, those benchmarks appear a bit easier.

Results obtained by **Qbfl** on those instances are sketched in Table 1. For each type of instance, the ratio of instances solved over all instances are shown in column "%solved" and the ratio of renamable Horn formulas reached over all checks performed are shown in the column "%RH". Those data are presented for both **Qbfl** with Jeroslow-Wang's and Δ heuristics.

The main observation is that really much more (about twice) benchmarks are solved by our solver using the heuristic Δ compared with Jeroslow-Wang's. The result is impressive, especially for *k_grz_n* instances: 30 benchmarks are solved with Jeroslow-Wang's heuristic while up to 59 are solved with Δ. Unfortunately, it is difficult to know if that good behavior is related to the renamable Horn detection or if the change of heuristics is alone responsible of it. Indeed, we have no way to compare the number of renamable Horn formulas found during the search by the two heuristics: reducing the search space also reduces the number of renamable Horn formulas found.

We tried Δ on some other types of formulas. In the crowd of existing benchmarks, we tried all instances proposed by A. Ayari (72), C. Castellini (169), M. Mneimneh and K. A. Sakallah (202), J. Rintanen (67) and C. Scholl and B. Becker (64)[6]. The main observation is that Δ does not alter significantly the

[5] Hard instances of this evaluation are those solved by one solver or even none.

[6] All those benchmarks are available on http://www.qbflib.org.

Table 1. Percents of benchmarks solved and renamable Horn instances reached

Instance	Jeroslow-Wang		Renamable Horn Δ		Instance	Jeroslow-Wang		Renamable Horn Δ	
type	%solved	%RH	%solved	%RH	type	%solved	%RH	%solved	%RH
k_branch_n	4.76	**25.26**	**9.52**	9.29	k_branch_p	4.76	**24.56**	4.76	9.33
k_d4_n	4.76	**13.25**	4.76	5.63	k_d4_p	9.52	**26.33**	**14.28**	25.34
k_dum_n	4.76	**11.69**	**23.80**	11.51	k_dum_p	4.76	**11.71**	**14.28**	11.40
k_grz_n	0	-	**61.90**	**11.94**	k_grz_p	0	-	0	-
k_lin_n	9.52	20.00	9.52	**24.26**	k_lin_p	9.52	**11.88**	**19.04**	8.65
k_path_n	9.52	5.47	**14.28**	**12.12**	k_path_p	14.28	7.43	**19.04**	**7.64**
k_ph_n	23.80	2.66	23.80	**11.73**	k_ph_p	19.04	**10.06**	19.04	6.89
k_poly_n	9.52	0	**14.28**	**6.66**	k_poly_p	4.76	**12.91**	**9.52**	11.04
k_t4p_n	4.76	11.37	4.76	**26.62**	k_t4p_p	0	-	**4.76**	**26.02**
Total valid	7.93	11.21	**18.51**	**13.30**	Total false	7.40	**14.98**	**11.64**	13.28

Instance	Jeroslow-Wang		Renamable Horn Δ	
type	%solved	%RH	%solved	%RH
Total valid	7.93	11.21	**18.51**	**13.30**
Total false	7.40	**14.98**	**11.64**	13.28
Total	7.67	13.09	**15.07**	**13.29**

behavior of **Qbfl** without it: Among the 283 instances solved using the Jeroslow-Wang heuristics, only 14 instances were not solved using Δ. Taking into account the Pan benchmarks, **Qbfl** equipped with Δ perform better than without it.

5 Conclusion

Our work focuses on the practical use of tractable restrictions of $\mathcal{QBF}$ in solvers. We proposed a heuristics based on Hébrard's algorithm to detect Horn renamability. We noticed that in practice current QBF solvers are not able to solve in reasonable time some instances of $\mathcal{QHF}$ or ren$\mathcal{QHF}$. Quantor[23], which is a solver somehow different from the other QBF solvers, performs also badly on those benchmarks.

In contrast, we tried some classical SAT solvers on the very same benchmarks (without the prefix): They can solve them easily. An explanation is that the extra complexity in the QBF case is due to the constraint on the prefix that most solvers try to satisfy first. We are currently investigating the behavior of Audemard and Saïs solver [14] on those benchmarks: Their solver considers last that constraint. Unfortunately, first experimental results show that even this solver can't easily solve those instances.

We are waiting for the results of the 2005 $\mathcal{QBF}$ evaluation to gather additional information that could help us to explain the behavior of the tested solvers.

References

1. Egly, U., Eiter, T., Tompits, H., Woltran, S.: Solving advanced reasoning tasks using quantified boolean formulas. In: AAAI'00, Austin (USA) (2000) 417–422
2. Fargier, H., Lang, J., Marquis, P.: Propositional Logic and One-stage Decision Making. In: KR'00, Breckenridge (CO) (2000) 445–456

3. Besnard, P., Schaub, T., Tompits, H., Woltran, S.: Paraconsistent Reasoning via Quantified Boolean Formulas, I: Axiomatising Signed Systems. In: JELIA'02, Cosenza (Italy) (2002) 320–331
4. Rintanen, J.: Constructing Conditional Plans by a Theorem-Prover. Journal of Artificial Intelligence Research **10** (1999) 323–352
5. Pan, G., Sattler, U., Vardi, M.Y.: BDD-Based Decision Procedures for K. In: CADE'02, Copenhagen (Denmark) (2002) 16–30
6. Cadoli, M., Giovanardi, A., Schaerf, M.: An algorithm to Evaluate Quantified Boolean Formulae. In: AAAI'98, Madison (USA) (1998) 262–267
7. Rintanen, J.: Improvements to the Evaluation of Quantified Boolean Formulae. In: IJCAI'99, Stockholm (Sweden) (1999) 1192–1197
8. Feldmann, R., Monien, B., Schamberger, S.: A distributed algorithm to evaluate quantified boolean formula. In: AAAI'00, Austin (USA) (2000) 285–290
9. Rintanen, J.: Partial implicit unfolding in the Davis-Putnam procedure for Quantified Boolean Formulae. In: QBF'01, Siena (Italy) (2001) 84–93
10. Giunchiglia, E., Narizzano, M., Tacchella, A.: Backjumping for Quantified Boolean Logic Satisfiability. In: IJCAI'01, Seattle (USA) (2001) 275–281
11. Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In: TABLEAUX'02, Copenhagen (Denmark) (2002)
12. Zhang, L., Malik, S.: Towards a symetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In: CP'02, Ithaca (USA) (2002) 200–215
13. Pan, G., Vardi, M.: Optimizing a BDD-Based Modal Solver. In: CADE'03, Miami Beach (USA) (2003) 75–89
14. Audemard, G., Saïs, L.: SAT based BDD solver for Quantified Boolean Formulas. In: ICTAI'04, Boca Raton (USA) (2004) 82–89
15. Aspvall, B., Plass, M., Tarjan, R.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Information Processing Letters **8** (1979) 121–123 Erratum: Information Processing Letters 14(4): 195 (1982).
16. Gent, I.P., Rowley, A.: Solving 2-CNF Quantified Boolean Formulae using Variable Assignment and Propagation. In: QBF'02, Cincinnati (USA) (2002) 17–25
17. Kleine-Büning, H., Karpinski, M., Flögel, A.: Resolution for quantified boolean formulas. Information and Computation **117** (1995) 12–18
18. Hébrard, J.J.: A Linear Algorithm for Renaming a Set of Clauses as a Horn Set. In: Theoretical Computer Science. Volume 124. (1994) 343–350
19. Jeroslow, R.J., Wang, J.: Solving propositional satisfiability problems. Annals of Mathematics and Artificial Intelligence **1** (1990) 167–188
20. Freemann, J.W.: Improvement to Propositional Satisfiability Search Algorithms. PhD thesis, University of Pennsylvania (1995)
21. Le Berre, D., Simon, L., Tacchella, A.: Challenges in the QBF Arena: the SAT'03 evaluation of QBF solvers. In: QBF'03, S. Margherita (Italy) (2003) 468–485
22. Balsiger, P., Heuerding, A., Schwendimann, S.: A benchmark method for the propositional modal logics K, KT, S4. Automated Reasoning **24** (2000) 297–317
23. Biere, A.: Resolve and Expand. In: SAT'04, Vancouver (Canada) (2004) 238–246

An Improved Upper Bound for SAT

Evgeny Dantsin and Alexander Wolpert

Roosevelt University,
430 S. Michigan Av.,
Chicago, IL 60605, USA
{edantsin, awolpert}@roosevelt.edu

Abstract. We give a randomized algorithm for testing satisfiability of Boolean formulas in conjunctive normal form with no restriction on clause length. Its running time is at most $2^{n(1-1/\alpha)}$ up to a polynomial factor, where $\alpha = \ln(m/n) + O(\ln\ln m)$ and n, m are respectively the number of variables and the number of clauses in the input formula. This bound is asymptotically better than the previously best known $2^{n(1-1/\log(2m))}$ bound for SAT.

1 Introduction

During the past few years there has been considerable progress in obtaining upper bounds on the complexity of solving the Boolean satisfiability problem. This line of research has produced new algorithms for k-SAT. They were further used to prove nontrivial upper bounds for SAT (no restriction on clause length).

Upper bounds for k-SAT. The best known upper bounds for k-SAT are based on two approaches: the satisfiability coding lemma [8, 7] and multistart random walk [10, 11]; both give close upper bounds on the time of solving k-SAT. The randomized algorithm in [10] has the $(2-2/k)^n$ bound where n is the number of variables in the input formula; the randomized algorithm in [7] has a slightly better bound. The multistart-random-walk approach is derandomized using covering codes in [2], which gives the best known $(2-2/(k+1))^n$ bound for deterministic k-SAT algorithms. For small values of k, these bounds are improved: for example, 3-SAT can be solved by a randomized algorithm with the 1.324^n bound [6] and by a deterministic algorithm with the 1.473^n bound [1].

Upper bounds for SAT (no restriction on clause length). The first nontrivial upper bound for SAT is given in [9]: the $2^{n(1-1/2\sqrt{n})}$ bound for a randomized algorithm based on the satisfiability coding lemma. A close bound for a deterministic algorithm is proved in [3]. A much better bound for SAT is the $2^{n(1-1/\log(2m))}$ bound, where m is the number of clauses in the input formula. This bound is due to Schuler [12] who gives a randomized algorithm that solves SAT using the k-SAT algorithm [8] as a subroutine. Schuler's algorithm is derandomized in [4]. The derandomization gives a deterministic algorithm that solves SAT with the same bound.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 400–407, 2005.

In this paper we improve the $2^{n(1-1/\log(2m))}$ bound: we give a randomized algorithm that solves SAT with the following upper bound on the running time:

$$2^{n\left(1-\frac{1}{\ln\left(\frac{m}{n}\right)+O(\ln\ln m)}\right)} \tag{1}$$

Idea of the algorithm. Our algorithm for SAT is basically a repetition of a polynomial-time procedure $\mathcal{P}$ that tests satisfiability of an input formula F. If F is satisfied by a truth assignment A, the procedure $\mathcal{P}$ finds A with probability at least p. As usual, repeating $\mathcal{P}$ on the input formula $O(1/p)$ times, we can find A with a constant probability.

When describing $\mathcal{P}$, we view clauses as sequences (rather than sets) of literals. We divide each clause into *blocks* of length k. More exactly, for a given integer $k \geq 1$, a clause $l_1, l_2, \ldots, l_s$ is divided into $b = \lceil s/k \rceil$ blocks as follows:

$$\boxed{l_1,\ldots,l_k,} \qquad \boxed{l_{k+1},\ldots,l_{2k},} \qquad \cdots \qquad \boxed{l_{k(b-1)+1},\ldots,l_s}$$

where each block (except the last one) consists of k literals. By the *first block* we mean the block consisting of $l_1, l_2, \ldots, l_k$. We say that a block is *true* under an assignment A if at least one literal in the block is true under A; otherwise we say that the block is *false* under A. If A is fixed, we omit the words "under A".

Let F consist of clauses $C_1, \ldots, C_m$; let A be a fixed satisfying assignment to F. The procedure $\mathcal{P}$ is based on the following dichotomy:

Case 1. The input formula F has "many" clauses in which the first block is false. We suppose that the number of such clauses is greater than or equal to some d. If we choose a clause C_i from $C_1, \ldots, C_m$ at random, the first block in C_i is false with probability at least d/m. Then we can simplify F by assigning "false" to all literals occurring in the first block of C_i.

Case 2. The input formula F has "few" clauses in which the first block is false. We suppose that the number of such clauses is less than d. Then we find A as follows. First, we guess those clauses in which the first block is false. Furthermore, we guess a true block in each such clause. Now we know a true block for each clause: it is either the first block or the block we have guessed. Let F' be the formula made up of these m true blocks. Obviously, F' is in k-CNF. Therefore we can use a k-SAT algorithm to find A.

This dichotomy suggests that $\mathcal{P}$ is a recursive procedure that invokes a subroutine $\mathcal{S}$ for processing Case 2. If the subroutine does not return a satisfying assignment, $\mathcal{P}$ simplifies the input formula (Case 1) and recursively invokes itself on the simplified formula. Both $\mathcal{P}$ and $\mathcal{S}$ use k and d as parameters.

Clearly, the success probability of $\mathcal{P}$ depends on values of the parameters k and d. What values of k and d maximize the success probability? We show that if we take $k \approx \log(m/n) + O(\log\log(m))$ and $d \approx n/\log^3 m$ then we obtain the following lower bound on the success probability:

$$2^{-n\left(1-\frac{1}{\ln\left(\frac{m}{n}\right)+O(\ln\ln m)}\right)}$$

In Sect. 2 we describe the procedure $\mathcal{P}$ and the subroutine $\mathcal{S}$. In Sect. 3 we define our algorithm for SAT and sketch a proof of bound (1) on its running time (see [5] for the full proof).

2 Procedure $\mathcal{P}$ and Subroutine $\mathcal{S}$

2.1 Notation

We deal with Boolean formulas in conjunctive normal form (CNF). By a *variable* we mean a Boolean variable that takes truth values t (true) or f (false). A *literal* is a variable x or its negation $\neg x$. If l is a literal then $\neg l$ denotes the complement literal, i.e. if l is x then $\neg l$ denotes $\neg x$, and if l is $\neg x$ then $\neg l$ denotes x. Similarly, if v denotes one of the truth values t or f, we write $\neg v$ to denote the complement truth value. A *clause* C is a sequence of literals such that C contains no complement literals. A *formula* F is a set of clauses; n and m denote, respectively, the number of variables and the number of clauses in F. If each clause in F contains at most k literals, we say that F is a *k-CNF formula.*

An *assignment* to variables $x_1, \ldots, x_n$ is a mapping from $\{x_1, \ldots, x_n\}$ to $\{\mathsf{t}, \mathsf{f}\}$. This mapping is extended to literals: each literal $\neg x_i$ is mapped to the truth value complement to the value assigned to x_i. We say that a clause C is *satisfied* by an assignment A (or, C is *true* under A) if A assigns t to at least one literal in C. Otherwise, we say that C is *falsified* by A (or, C is *false* under A). The formula F is *satisfied* by A if every clause in F is satisfied by A. In this case, A is called a *satisfying* assignment for F.

Let F be a formula and $l_1, \ldots, l_s$ be literals such that their variables occur in F. We write $F[l_1 = \mathsf{f}, \ldots, l_s = \mathsf{f}]$ to denote the formula obtained from F by assigning the value f to all of $l_1, \ldots, l_s$. This formula is obtained from F as follows: the clauses that contain any literal from $\neg l_1, \ldots, \neg l_s$ are deleted from F, and the literals $l_1, \ldots, l_s$ are deleted from the other clauses. Note that $F[l_1 = \mathsf{f}, \ldots, l_s = \mathsf{f}]$ may contain the empty clause or may be the empty formula.

Let A and A' be two assignments that differ only in the values assigned to a literal l. Then we say that A' is obtained from A by *flipping* the value of l.

By *SAT* we mean the following computational problem: Given a formula F in CNF, decide whether F is satisfiable. The *k-SAT* problem is the restricted version of SAT that allows only clauses consisting of at most k literals.

We write $\log x$ to denote $\log_2 x$.

2.2 Description of $\mathcal{S}$

Both procedures $\mathcal{S}$ and $\mathcal{P}$ use the parameters k and d; their values will be determined in the next section.

Suppose that an input formula F has a satisfying assignment such that F has only "few" ($< d$) clauses in which the first block is false under this assignment. Then the subroutine $\mathcal{S}$ finds such an assignment (with some probability estimated in Sect. 3). The subroutine takes two steps:

1. Reduction of F to a k-CNF formula F' such that any satisfying assignment to F' satisfies F.
2. Use of a k-SAT algorithm to find a satisfying assignment to F'.

At the first step, $\mathcal{S}$ guesses all "bad" clauses for some satisfying assignment A, i.e. clauses in which the first block is false under A. More exactly, the subroutine guesses a (possibly) larger set of clauses: a set $\{B_1, \ldots, B_{d-1}\}$ such that all "bad" clauses are contained in this set. For each clause B_i, the subroutine guesses a true block in B_i. Thus, the subroutine gets a true block for each clause in F – the guessed true blocks for $B_1, \ldots, B_{d-1}$ and the first blocks for the other clauses in F. These true blocks make up F'. It is obvious that A satisfies F'.

To test satisfiability of k-CNF formulas at the second step, $\mathcal{S}$ uses a randomized polynomial-time algorithm that finds a satisfying assignment with an exponentially small probability. We choose Schöning's algorithm [10] to perform this testing (we could choose any algorithm that has at least the same success probability, for example the algorithm [7] based on the satisfiability coding lemma). More exactly, we use "one random walk" of Schöning's algorithm, which has the success probability at least $(2 - 2/k)^{-n}$ up to a constant [11].

Note that if $d = 1$, i.e. there is no "bad" clause, then $\mathcal{S}$ simply finds a satisfying assignment to the k-CNF formula made up of the m first blocks. Also note that the smaller d, the higher the probability of guessing a formula consisting of true blocks.

Subroutine $\mathcal{S}$

```
Input: Formula $F$ with $m$ clauses over $n$ variables, integers $k$ and $d$.
Output: Satisfying assignment or "no".

1. Reduce $F$ to a $k$-CNF formula $F'$ as follows:
   (a) Choose $d-1$ clauses $B_1, \ldots, B_{d-1}$ in $F$ at random.
       Comment: Guess a set that contains all "bad" clauses.
   (b) For each $B_i$, choose a block in $B_i$ at random and replace $B_i$ by the chosen block.
       Comment: Guess a true block in each $B_i$ and replace $B_i$ by this true block.
   (c) Replace each clause not belonging to $\{B_1, \ldots, B_{d-1}\}$ by its first block.
       Comment: The first block in each "good" clause is assumed to be true.
2. Test satisfiability of $F'$ using one random walk of length $3n$ (see [10] for details):
   (a) Choose an initial assignment $a$ uniformly at random;
   (b) Repeat $3n$ times:
       i. If $F'$ is satisfied by the assignment $a$ then return $a$ and stop;
       ii. Pick any clause $C$ in $F'$ such that $C$ is falsified by $a$. Choose a literal $l$ in $C$ uniformly at random. Modify $a$ by flipping the value of $l$.
   (c) Return "no".
```

2.3 Description of $\mathcal{P}$

The procedure first calls Subroutine $\mathcal{S}$. The result of $\mathcal{S}$ depends on which case of the dichotomy holds.

1. For every satisfying assignment A (if any), the input formula F has "many" ($\geq d$) clauses in which the first block is false under A. Then the subroutine never finds A.
2. For a satisfying assignment A, the input formula F has "few" ($< d$) clauses in which the first block is false. Then the subroutine returns a satisfying assignment with its success probability.

The "no" answer from $\mathcal{S}$ is treated as Case 1 of the dichotomy. Therefore, the procedure $\mathcal{P}$ simplifies F and recursively invokes itself on the simplified formula. To simplify F, the procedure chooses a clause C at random from the "long" clauses in F, i.e. the clauses that have more than one block. Then $\mathcal{P}$ assigns f to all literals in the first block of C and reduces F to $F[l_1 = \mathsf{f}, \ldots, l_k = \mathsf{f}]$. Why can we restrict the choice to "long" clauses? Because if the input formula is satisfiable then all one-block clauses must be true.

Procedure $\mathcal{P}$

Input: Formula F with m clauses over n variables, integers k and d.
Output: Satisfying assignment or "no".

1. Invoke $\mathcal{S}$ on F, k, and d.
   Comment: If there are less than d "bad" clauses, the subroutine returns a satisfying assignment (with its success probability) and stops.
2. Choose clause C at random from those clauses in F that have more than one block.
   Comment: We guess a "long" clause in which the first block is false.
3. Simplify F by assigning f to all literals of the first block in C. Namely, if the first block of C consists of $l_1, \ldots, l_k$, reduce F to $F[l_1 = \mathsf{f}, \ldots, l_k = \mathsf{f}]$.
   Comment: We simplify F by eliminating the guessed variables.
4. Recursively invoke $\mathcal{P}$ on $F[l_1 = \mathsf{f}, \ldots, l_k = \mathsf{f}]$.
5. Return "no".

Note that Schuler's algorithm [12] can be viewed as a special case of $\mathcal{P}$: take $d = 1$, $k = \log(2m)$, and use a different method for testing k-CNF formulas at step 2 in $\mathcal{S}$ (the algorithm based on the satisfiability coding lemma [8] instead of Schöning's algorithm).

3 Main Result

Given an input formula F and some values of k and d, Procedure $\mathcal{P}$ finds a fixed satisfying assignment A or returns "no". What is the probability of finding A? We prove the following lower bound on the success probability of $\mathcal{P}$:

$$\left(\frac{\sqrt{m}}{3e}\right) (emn)^{-(d-1)} \, 2^{-n\left(1 - \frac{\log e}{k}\right)} \tag{2}$$

where n and m are respectively the number of variables and the number of clauses in F. The proof is more or less straightforward, but technical. We omit it here due to space limitation, see [5] for the full proof.

What values of k and d maximize bound (2) for given n and m? The analysis in [5] shows that for n and m such that $10 \leq n < em \leq 2^{\sqrt[3]{n/2}}$, the maximum is attained when k and d are chosen as follows:

$$k_0 = \left\lceil \frac{\log\left(\frac{em}{n}\right) + 3\log\log(em)}{1 + \frac{\log e}{\log^3(em)}} \right\rceil \qquad d_0 = \left\lceil \frac{n}{\log^3(em)} \right\rceil \tag{3}$$

Using these values of the parameters, we define our main algorithm as a repetition of $\mathcal{P}$. The number r of repetitions is taken so that the success probability of $\mathcal{P}$ is amplified to a constant probability. Namely, we substitute k_0 and d_0 in lower bound (2) and take the inverse:

$$r = \left\lceil \left(\frac{3en}{\sqrt{m}}\right) 2^{n\left(1-\frac{\log e}{k_0+2}\right)} \right\rceil \tag{4}$$

Algorithm $\mathcal{A}$

Input: Formula F in CNF with m clauses over n variables.
Output: Satisfying assignment or “no”.

```
1. Compute $k_0$ and $d_0$ as in (3) and $r$ as in (4).
2. Repeat the following $r$ times:
   (a) Run $\mathcal{P}(F, k_0, d_0)$;
   (b) If a satisfying assignment is found, return it and stop.
3. Return “no”.
```

Theorem 1. *Algorithm $\mathcal{A}$ runs in time*

$$O(n^3 m)\ 2^{n\left(1-\frac{\log e}{k_0+2}\right)}$$

For any satisfiable input formula such that $10 \leq n < em \leq 2^{\sqrt[3]{n/2}}$, Algorithm $\mathcal{A}$ finds a satisfying assignment with probability greater than $1/2$.

Proof (see [5] for details). To prove the first claim, we need to estimate the running time of $\mathcal{P}$. The procedure recursively invokes itself at most $\lfloor n/k_0 \rfloor$ times. Each call scans the input formula and eliminates at most k_0 variables. Therefore, the procedure performs $O(n)$ scans. Algorithm $\mathcal{A}$ repeats $\mathcal{P}$ at most r times, which gives the bound in the claim.

The success probability of $\mathcal{A}$ is at least $1 - (1 - q(n, m))^r$ where $q(n, m)$ is the result of substituting k_0 and d_0 in lower bound (2), i.e. $q(n, m)$ is a lower bound on the success probability of $\mathcal{P}$ for n, m, k_0, and d_0. Then the second claim follows from the inequality $(1 - x)^r \leq e^{-xr}$. □

Remark 1. Note that we can write the bound in Theorem 1 as

$$O(n^3 m)\ 2^{n\left(1-\frac{1}{\ln\left(\frac{m}{n}\right)+O(\ln\ln m)}\right)}.$$

Remark 2. Our algorithm can be viewed as a generalization of Schuler's algorithm [12]. The latter is derandomized with the same upper bound on the running time [4]. It would be natural to try to apply the same method of derandomization to our algorithm. However, the direct application gives a deterministic algorithm with a much worse upper bound. Is it possible to derandomize Algorithm $\mathcal{A}$ with the same or a slightly worse upper bound?

References

1. T. Brueggemann and W. Kern. An improved local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1-3):303–313, December 2004.
2. E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.
3. E. Dantsin, E. A. Hirsch, and A. Wolpert. Algorithms for SAT based on search in Hamming balls. In *Proceedings of the 21st Annual Symposium on Theoretical Aspects of Computer Science, STACS 2004*, volume 2996 of *Lecture Notes in Computer Science*, pages 141–151. Springer, March 2004.
4. E. Dantsin and A. Wolpert. Derandomization of Schuler's algorithm for SAT. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, SAT 2004*, pages 69–75, May 2004.
5. E. Dantsin and A. Wolpert. An improved upper bound for SAT. *Electronic Colloquium on Computational Complexity*, Report TR05-030, March 2005.
6. K. Iwama and S. Tamaki. Improved upper bounds for 3-SAT. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, page 328, January 2004. A preliminary version appeared in Electronic Colloquium on Computational Complexity, Report No. 53, July 2003.
7. R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, FOCS'98*, pages 628–637, 1998.
8. R. Paturi, P. Pudlák, and F. Zane. Satisfiability coding lemma. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS'97*, pages 566–574, 1997.
9. P. Pudlák. Satisfiability – algorithms and logic. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98)*, volume 1450 of *Lecture Notes in Computer Science*, pages 129–141. Springer-Verlag, 1998.
10. U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, FOCS'99*, pages 410–414, 1999.

11. U. Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002.
12. R. Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms*, 54(1):40–44, January 2005. A preliminary version appeared as a technical report in 2003.

Bounded Model Checking with QBF

Nachum Dershowitz[1], Ziyad Hanna[2], and Jacob Katz[2]

[1] School of Computer Science., Tel-Aviv University, Israel
nachumd@cs.tau.ac.il
[2] Intel Corporation, Haifa, Israel
{ziyad.hanna,jacob.katz}@intel.com

Abstract. Current algorithms for bounded model checking (BMC) use SAT methods for checking satisfiability of Boolean formulas. These BMC methods suffer from a potential memory explosion problem. Methods based on the validity of Quantified Boolean Formulas (QBF) allow an exponentially more succinct representation of the checked formulas, but have not been widely used, because of the lack of an efficient decision procedure for QBF. We evaluate the usage of QBF in BMC, using general-purpose SAT and QBF solvers. We also present a special-purpose decision procedure for QBF used in BMC, and compare our technique with the methods using general-purpose SAT and QBF solvers on real-life industrial benchmarks. Our procedure performs much better for BMC than the general-purpose QBF solvers, without incurring the space overhead of propositional SAT.

1 Introduction[1]

Model checking is a technique for the verification of the correctness of a finite-state system with respect to a desired behavior. The system is traditionally modeled as a labeled state-transition graph, and the behavior is specified by a temporal logic formula. Early implementations, based on explicit-state model checking, suffered from the state explosion problem. The introduction of symbolic model checking with binary decision diagrams (BDDs) and other recently developed methods, such as Bounded Model Checking (BMC), succeeded in partially overcoming this problem and enabled industrial applications of model checking for real-life systems, mostly in the hardware design industry. However, all these methods still suffer from the potential memory explosion problem on modern test cases. In this work we evaluate the application of Quantified Boolean Formulas (QBF) in BMC of safety properties in attempt to avoid the memory explosion problem. We also present a special-purpose purpose QBF decision procedure for a QBF encoding of BMC problems.

Assume a system M=(S, I, TR), where S is the set of states, I is the characteristic function of the set of the initial states, and TR is the transition relation. Let F be a characteristic function of the "bad" states violating the property being checked. As in classical BMC, the fact that a "bad" state Z_k is reachable from an initial state Z_0 in exactly k steps may be formulated by "unrolling" the transition relation k times:

[1] Due to space constraints references have been omitted in this text.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 408 – 414, 2005.

$$R_k(Z_0,Z_k) = \exists Z_1,...,Z_{k-1} : I(Z_0) \wedge F(Z_k) \wedge \bigwedge_{i=0}^{k-1} TR(Z_i,Z_{i+1}). \tag{1}$$

The validity of this formula may be proved or disproved by applying a SAT decision procedure on its propositional part. Noticeably, the number of copies of the transition relation *TR* in this formula is the same as the number of steps being checked. When iteratively increasing the bound *k*, each successive iteration checks reachability of the final states in one more step than the previous iteration. Thus, for a complete check, the SAT procedure must be invoked on formulas containing an exponential number of copies of the transition relation.

To partially overcome the potential memory explosion, the following QBF formulation of the bounded reachability problem can be used:

$$\begin{aligned} &R_k(Z_0,Z_k) = \exists Z_1,...,Z_{k-1} : I(Z_0) \wedge F(Z_k) \wedge \\ &\forall U,V : \left(\bigvee_{i=0}^{k-1} (U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1}) \right) \rightarrow TR(U,V). \end{aligned} \tag{2}$$

The formula (2) contains only one copy of the transition relation. Increasing the bound, thus, would mean an addition of a new intermediate state and a term of the form $(U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1})$. Hence, the formula increase from iteration to iteration does not depend on the size of the transition relation, which is usually the biggest formula in the specification of the model.

2 jSAT Decision Procedure

An experimental evaluation of general-purpose QBF solvers on formulas of form (2), presented in section 3, found them very inefficient, as they failed to solve practically any of the formulas in our test bench. This fact motivated the development of a special-purpose decision procedure, called jSAT, for formulas of this form.

jSAT holds in memory the encoding variables representing the states Z_0, Z_1, …, Z_k, *U* and *V*, but only holds the following propositional formula:

$$I(Z_0) \wedge TR(U,V) \wedge F(Z_k). \tag{3}$$

The states Z_i ($0 \leq i \leq k$) represent a path; the states *U* and *V* represent two neighboring states in that path. Instead of explicitly storing the fact that *U* and *V* represent a pair of neighboring states, as done in (2) with assistance of the terms of the form $(U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1})$, our algorithm implicitly assumes this information.

jSAT is based on the classic DPLL algorithm widely used in the current state-of-the-art SAT and QBF solvers. Intuitively, jSAT algorithm can be seen as a depth-first search in the state graph of the system from the initial states to the final ones. The algorithm starts by associating *U* with Z_0 and *V* with Z_1; thus the formula (3) becomes semantically equivalent to:

$$I(Z_0) \wedge TR(Z_0,Z_1) \wedge F(Z_k). \tag{4}$$

```
jSAT() {
    InitializeCurrentAndNextStates();
    while (true) {
        if (! SelectDecisionVariable()) {
            if (AllStatesDecided()) return true;
            if (! AdvanceCurrentState()) return false;
        }
        while (! BCP()) {
            if (! ResolveConflict()) return false;
        }
    }
}
```

Fig. 1. Pseudo-code of jSAT decision procedure

The states Z_0 and Z_1 are then chosen by finding an assignment to their encoding variables, if possible, so that Z_0 is an initial state and Z_1 is its successor. As soon as they are chosen, the algorithm makes Z_1 to be the current state and Z_2 to be the next one: U becomes an alias to $Z_{1,}$ and V becomes an alias to $Z_{2.}$ The algorithm proceeds in this fashion until all states are successfully chosen, or until it discovers that such a choice is impossible.

The pseudo-code of the algorithm is shown on Fig. 1. The algorithm first initializes the states U and V to be associated with Z_0 and Z_1, respectively. The procedure SelectDecisionVariable() selects a still unassigned variable out of the encoding variables of the current state or, if all the encoding variables of the current state are assigned, from those of the next state. We restrict the decision strategy to selecting decision variables in the order of the states in the path: encoding variables of the state Z_0 are selected first, then the variables of Z_1, then the variables of Z_2, and so on. Such a restriction causes the algorithm to implement a depth-first search of the state graph and to "visit" only the states actually reachable from the initial states. The order of the selection of the encoding variables within one state is not important, and heuristics similar to the ones used in SAT/QBF solvers can be used.

SelectDecisionVariable() returns true if the decision is made successfully. Boolean Constraint Propagation is then performed by the procedure BCP(), which returns false in case of a conflict. If a conflict is produced, ResolveConflict() attempts to analyze it and backtrack to a previous decision level. In case the conflict cannot be resolved the algorithm terminates and the given formula is reported invalid.

SelectDecisionVariable() returns false whenever all the encoding variables of the current and the next states have been decided. If at this point all the states have been decided, as determined by the call to AllStatesDecided(), then a path has been found from an initial state to a final one, and the algorithm terminates, reporting the given formula is valid. Otherwise, if undecided states remain, AdvanceCurrentState()

```
ResolveConflict() {
    nBacktrackingLevel = AnalyzeConflict();
    if (nBacktrackingLevel < 0) return false;
    nFirstUndecidedPathState = Backtrack(nBacktrackingLevel);
    if (! RetractCurrentAndNextStates(nFirstUndecidedState))
        return false;
    return true;
}
```

Fig. 2. Pseudo-code of jSAT decision procedure

advances *U* and *V* to the next pair of states by associating *U* with whatever was previously associated with *V*, and associating *V* with the next state in the path. During this operation new relations between the encoding variables become apparent. Thus, for example, when *U* and *V* are moved from the pair of states (Z_0, Z_1) to the next pair (Z_1, Z_2), the relations between the encoding variables of Z_1 and Z_2 become explicit in *TR*(*U*, *V*). Since the newly discovered information may contradict some of the already made decisions, conflicts may arise during the adjustment operations. The procedure AdvanceCurrentState() returns false in case a conflict occurred that could not be resolved; in this case the algorithm terminates and the given formula is invalid.

Fig. 2 shows the pseudo-code of ResolveConflict() procedure. The call to AnalyzeConflict() checks whether the conflict is resolvable, and if yes, produces a conflict clause and returns the decision level to which to backtrack. Then, by the call to Backtrack(), the algorithm undoes the assignments made on the decision levels higher than the level to which the algorithm should backtrack. Backtrack() returns the earliest state among all the states, which does not have all its encoding variables assigned after the backtracking. If this earliest state is the one currently associated with *U* (i.e. is the current state) or an earlier one, *U* and *V* are retracted by RetractCurrentAndNextStates(), so that *V* is associated with the earliest undecided state in the path. This retraction implements the retreating step of the depth-first search in the state graph, so that the search is directed into another part of the graph. Noticeably, as with the operation of advancement of the current and the next states, the retraction may also produce conflicts, because the relations that were not explicit in the formula become explicit. RetractCurrentAndNextStates() returns false in case an irresolvable conflict occurred during the operation.

An important aspect of our algorithm follows from the fact that *U* and *V* represent different states at different points of time. It is therefore generally incorrect to produce learned conflict clauses that involve the encoding variables of *U* or *V,* or any artificial variable resulting from the translation of *TR*(*U*, *V*) to CNF, as they will become useless as soon as *U* and *V* are adjusted to represent another pair of states. Therefore, the learned clauses must be formulated in terms of the encoding variables of Z_i. Our conflict analysis technique achieves this by using only decision variables in the learned clauses, somewhat similar to Last UIP learning scheme described in.

3 Experimental Results

We have implemented jSAT algorithm to measure its applicability to the problem of BMC. We used a bounded model checker to generate formulas of the forms (1), (2) and (3). The formulas of form (1) were generated in DIMACS format and could be fed into many available SAT solvers. The formulas of forms (2) were generated in QDIMACS format and could be fed into the available QBF solvers. The formulas of form (3) were generated in a slightly customized DIMACS format, which adds the specification of the encoding variables to the formula description; our implementation of jSAT reads this modified DIMACS format.

Table 1. Number of instances solved by each solver per test case. There is a total of 18 instances in each test case, corresponding to BMC problems with bounds 3 to 20. SAT and UNSAT instances are shown separately. '-' sign specifies that there are no instances with the specific result for the corresponding test case

	# vars	jSat		Base		zChaff	
		SAT	*UNSAT*	*SAT*	*UNSAT*	*SAT*	*UNSAT*
test08	10	-	16	-	18	-	18
test12	11	18	-	18	-	18	-
test10	12	-	18	-	18	-	18
test03	39	18	-	18	-	18	-
test06	160	-	1	-	12	-	18
test09	160	18	-	18	-	18	-
test05	199	-	0	-	18	-	18
test11	220	14	4	14	4	14	4
test04	626	0	1	4	2	13	2
test13	662	18	-	18	-	18	-
test02	914	-	0	-	13	-	18
test07	1055	0	-	11	-	17	-
test01	2013	18	-	5	-	11	-
Total (out of 234)		104	40	106	85	127	96
		144		191		223	

We used a set of thirteen proprietary Intel® model checking test cases of different sizes to compare the run-time and memory consumption of the different BMC methods. For each test case we generated formulas of all kinds for the bounds in range from 3 to 20, resulting in the total amount of 234 formulas of each kind. Some of the formulas of form (3) were publicly disclosed and participated in the QBF solver evaluation during SAT2004 conference. We used a dual Intel® Xeon™ 2.8 GHz Linux RH7.1 workstation with 4GB of memory for the experiments, and set a 600 second time out and 1 GB memory limits on all solvers.

We first used QuBE state-of-the-art QBF solver to solve formulas of form (2) for all the test cases and to compare its run-time to the run-time of SAT solvers on the corresponding instances of form (1). We discovered that QuBE was able to solve only a few of the 234 formulas within the set time limits. This fact served as our motivation for the development of jSAT. We expected that jSAT, as a special-purpose decision procedure, would demonstrate memory consumption as low as the general-purpose QBF solvers, but a better run-time. We did not expect that jSAT run-time would be as good as that of the SAT solvers on the corresponding instances.

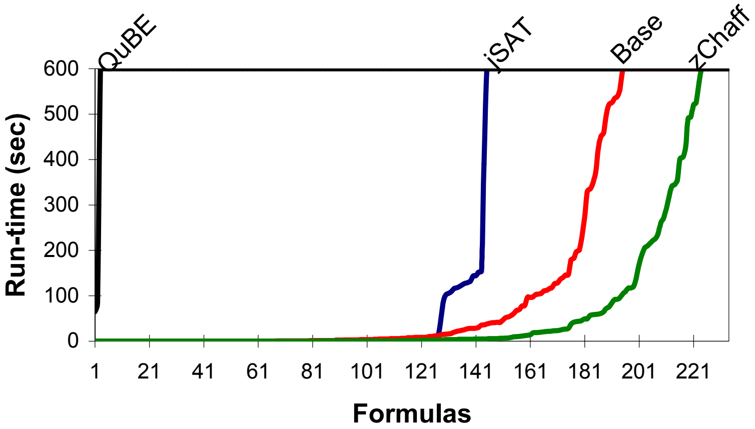


Fig. 3. Number of instances solved by each solver vs. the CPU time consumed

Our implementation is based on an existing solver[2] (base solver), which is reported to have slightly slower performance than zChaff. To perform a fair analysis we chose to compare our algorithm to that base solver, but also provide a comparison to zChaff II, as one of the best-known state-of-the-art solvers.

Table 1 shows the sizes of the test cases in terms of the state variables in the model, and the number of formulas each of the solvers successfully coped with (QuBE has been omitted in this table for brevity). The numbers of solved SAT and UNSAT instances are shown separately. (We use the terms "SAT" and "UNSAT" in case of jSAT for consistency, even though jSAT solves a QBF. SAT result in this case means that the instance was proved valid; UNSAT means it was proved invalid.) Interestingly, jSAT's results are especially close to those of the base solver on SAT instances, where jSAT managed to solve 104 versus 106 instances solved by the base solver.

On UNSAT instances, the distance between jSAT and the base solver is much more significant. Fig. 3 graphically shows the run-time performance of the solvers. The x-axis shows the number of instances solved, and y-axis shows the time taken to solve a particular instance; the curve is obtained by sorting the run-times in an ascending order. It is evident that jSAT significantly outperformed the general-purpose QBF solver QuBE. It still did not achieve the same run-times as the SAT solvers, though in the biggest test case test01 (see Table 1) it managed to solve in seconds all the instances, which required a much longer time for the other solvers. Also it is no-

[2] Y. Feldman, N. Dershowitz, Z. Hanna. "Parallel Multithreaded Satisfiability Solver: Design and Implementation". Workshop on Parallel and Distributed Model Checking (PDMC), 2004.

ticeable that on most of the instances that jSAT succeeded to solve the run-time achieved by jSAT is similar to that of the SAT solvers. However, jSAT performance degrades much faster than that of the SAT solvers when coming to the more complex instances: the slope of the performance curve of jSAT is much higher than that of the other solvers.

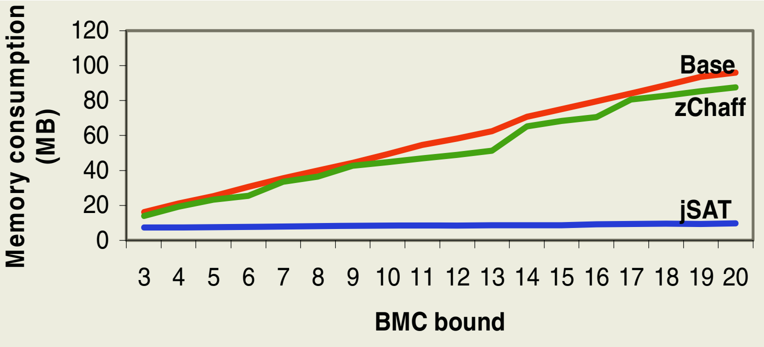


Fig. 4. Memory consumption of each solver on the instances generated for the test case test13

Fig. 4 graphically shows the memory consumed by jSAT, the base solver and zChaff when solving instances generated from the test case test13, which is the largest test case fully solved by all the three tools. The x-axis shows the BMC bound, and the y-axis shows the memory consumed when solving the corresponding instance. The run-time of jSAT on these instances varied from 1 to 3 seconds; the run-time of zChaff 1 to 6 seconds; and the run-time of the base solver from 3 to 146 seconds. As expected, the graph indicates that jSAT memory consumption practically does not depend on the BMC bound being solved, while for SAT-based BMC approaches the memory consumption is proportional to the bound. The same behavior has been observed on the other test cases, including those that jSAT fails to complete.

4 Conclusions

We have presented an evaluation of the usage of QBF in BMC, comparing a classical SAT-based BMC method to one using a QBF encoding of the problem, which avoids the memory explosion problem because it does not require the “unrolling” of the transition relation. We found that modern state-of-the-art general-purpose QBF solvers are still unable to handle the real-life instances of BMC problems in an efficient manner.

As the main contribution of our work, we presented a special-purpose QBF decision procedure for the solution of QBF instances encoding BMC problems in form (2). A performance evaluation of our algorithm shows that it achieves the expected memory savings, and succeeds to solve significantly more instances than a general-purpose QBF solver. Still, jSAT does not achieve run-times as short as the state-of-the-art SAT solvers on the corresponding SAT instances of the same problems, even though on some benchmarks it shows similar, and sometimes better, run-times.

Variable Ordering for Efficient SAT Search by Analyzing Constraint-Variable Dependencies

Vijay Durairaj and Priyank Kalla

Department of Electrical and Computer Engineering,
University of Utah, Salt Lake City, UT-84112
{durairaj, kalla}@ece.utah.edu

Abstract. This paper presents a new technique to derive an initial static variable ordering for efficient SAT search. Our approach not only exploits variable activity and connectivity information simultaneously, but it also analyzes how tightly the variables are related to each other. For this purpose, a new metric is proposed - the degree of correlation among pairs of variables. Variable activity and correlation information is modeled (implicitly) as a weighted graph. A topological analysis of this graph generates an order for SAT search. Also, the effect of decision-assignments on clause-variable dependencies is taken into account during this analysis. An algorithm called ACCORD (ACtivity - CORrelation - ORDering) is proposed for this purpose. Using efficient implementations of the above, experiments are conducted over a wide range of benchmarks. The results demonstrate that: (i) the variable order generated by our approach significantly improves the performance of SAT solvers; (ii) time to derive this order is a fraction of the overall solving time. As a result, our approach delivers faster performance as compared to contemporary approaches.

1 Introduction

An important aspect of CNF-SAT is to derive an ordering of variables to guide the search. The order in which variables (and correspondingly, constraints) are resolved significantly impacts the performance of SAT search procedures. **Variable activity** and **clause connectivity** are often considered as qualitative and quantitative metrics to model clause-variable dependencies. Activity of a variable (or literal) is defined as the number of its occurrence among all the clauses of a given SAT problem [1]. Most conventional SAT solvers [2] [3] [4] employ variable/literal-activity based branching heuristics to resolve the constraints.

Connectivity of constraints has also been used as a heuristic approach to derive variable orderings for SAT search. Loosely speaking, two clauses are said to be "connected" if one or more variables are common to their support. Clause connectivity can be modeled by representing CNF-SAT constraints as (hyper-) graphs and, subsequently, analyzing the graph's topological structure. Tree decomposition techniques have been proposed in literature [5] [6] for analyzing connectivity of constraints in constraint satisfaction programs (CSP). Such techniques identify decompositions with **minimum tree-width**, thus enabling

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 415–422, 2005.

a partitioning of the overall problem into a chain of connected constraints. Recently, such approaches have also found application in those problems that can be modeled as DPLL-based CNF-SAT search [6] [7]. Various approaches operate on such partitioned tree structures by deriving an order in which the partitioned set of constraints are resolved [6] [7] [8] [9] [10] [7] [11] [10]. Recently, Durairaj *et. al.* [12] proposed hypergraph bi-partitioning based constraint decomposition scheme (HGPART) that employs both variable activity and clause connectivity simultaneously to derive a variable order.

The above connectivity/tree-decomposition/partitioning-based methods that guide SAT diagnosis have one or more of the following limitations: (i) they suffer from large compute times to search the variable order [5] [9] [6]; (ii) the quality of the variable order does not consistently improve the performance of SAT solvers [9] [11]; (iii) there is no direct control over the decomposition [12]. Another limitation of these techniques is that while they do analyze variable activity, clause connectivity or both, however, they do not analyze how tightly the variables are connected to each other. As a result, tightly connected, hard problems may not realize the run-time improvements.

To overcome the above limitations , this paper presents a new approach to derive an initial static ordering for SAT search by rigorously analyzing constraint-variable dependencies. Moreover, we analyze the effect of decision-assignments on the variable order and exploit this effect to further improve the order. Experimental results demonstrate that our approach is faster and more robust than the contemporary variable ordering techniques, and it improves the performance of SAT solvers (in many cases by orders of magnitude).

2 ACCORD: Activity-Correlation Based Ordering

It is our desire to derive a variable order for SAT search by analyzing clause-variable relationships. The importance of branching on high activity variables is well understood [1]. Analyzing the connectivity of constraints is also important for constraint resolution. Contemporary techniques address both of the above issues. However, 'how tightly are the variables related?' - this feature too should not be overlooked. For this purpose, we propose a metric that measures how tightly the variables are connected/related. We define this metric as follows:

Definition 2.1. *Two variables x_i and x_j are said to be* **correlated** *if they appear together (as literals) in one or more clauses. The number of clauses in which the pair of variables (x_i, x_j) appear together is termed as their* **degree of correlation**.

In our approach, the constraint-variable relationship of a given CNF-SAT problem is modeled as a weighted graph. The variables (as opposed to literals) form the vertices, while edges denote the correlation/connectivity between them. Associated with each variable is its activity, which is modeled as an integer value within the node. The edge weights represent the degree of correlation between the variables/vertices. For example, if two variables x_i, x_j appear together in n

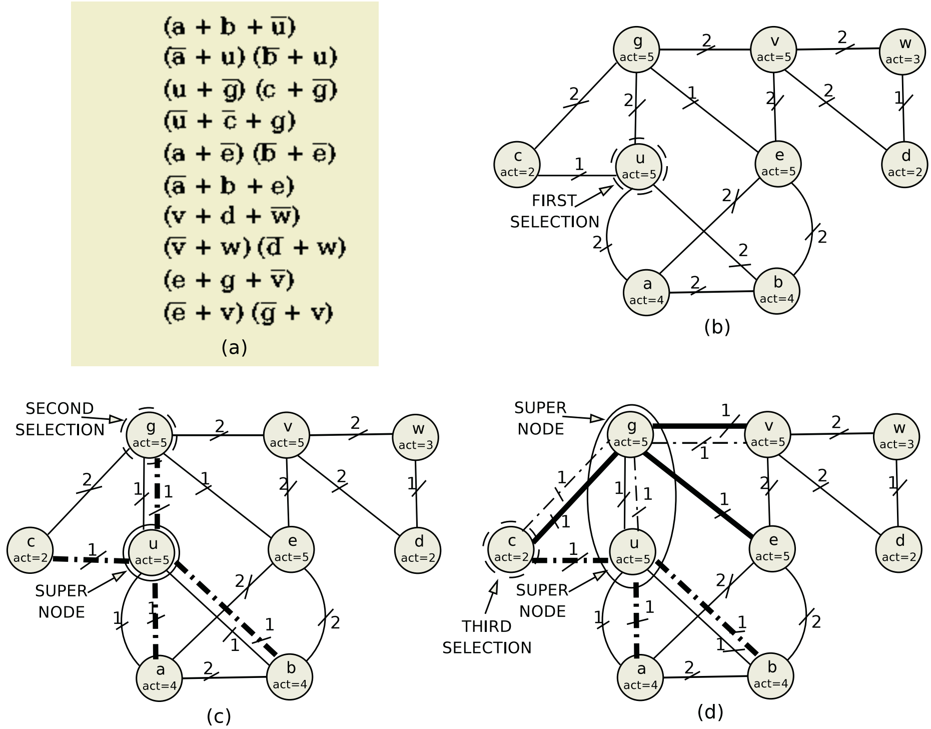


Fig. 1. An example CNF and its Weighted Graph - Edge weights denote the degree of correlation between the variables (vertices). Variable activity depicted within the node

clauses, then the weight of the edge e_{ij} connecting them is n. An ordering of the nodes (variable order) can be performed by analyzing the graph's topology. We now describe our approach by means of an example corresponding to the CNF-SAT problem shown in Fig. 1(a). Its corresponding weighted graph is depicted in Fig. 1(b).

We begin the search by first selecting the highest active variable, i.e. the node that has the highest internal weight. The variable is marked and the node is added to a set called *supernode*. The variable is also stored in a list (*var_ord_list*). The SAT tool should branch on this variable first. It can be observed from the graph (weight within the nodes) that the activity of variables $\{e, u, g, v\}$ is the highest ($= 5$). We select one of these variables (randomly or lexicographically) as the initial branching variable. Let us select the variable u, shown in Fig. 1(b) as "First Selection." The variable u becomes the supernode. Now we need to identify the set of variables connected to this supernode u. Note that, when the solver branches on this variable u, it will assign $u = 1$. This is because the activity of its positive literal is greater than that of negative. Hence, all the clauses corresponding to the literal u will be satisfied due to this assignment. In order to exploit this behaviour, our algorithm determines connectivity only from the clauses in which literal $\bar{u}$ appears. For this purpose, the incident edges on the node u are split into two edges corresponding to the literals and their correlation as shown in Fig. 1(c). The dotted edge corresponds to the negative literal ($\bar{u}$). All the nodes that share at least one edge with the supernode ($\bar{u}$) are identified. The nodes $\{g, c, a, b\}$ are connected to the literal $\bar{u}$, as shown in Fig. 1(c). One of these variables $\{g, c, a, b\}$ is to be selected as the next branching variable. We consider the degree of correlation, modeled as edge weights, as the metric to identify this variable. The nodes that share an edge with the supernode are sorted in *decreasing order of the sum of their degrees of correlation with the nodes in the supernode.* In our example, the variables $\{g, a, b, c\}$ have the same degree of correlation with node u, which is equal to 1. In order to break this tie, *we further distinguish these variables according to their activity.* Since the activity of variable g is higher than that of variables $\{a, b, c\}$, g is selected as the next branching variable. This is shown in Fig. 1(c) as "Second Selection." Moreover, the node g is added to the current supernode set; *supernode* $= \{u, g\}$. The variable order derived so far is *var_ord_list* $= \{u, g\}$.

Algorithm 1. Pseudo code for ACCORD

```
INPUT = CNF Clauses and Variable-Clause Database ;
/* For each variable $x_i$, the Variable-Clause database contains a list of clauses in
   which $x_i$ appears */
OUTPUT = Variable order for SAT search ;
activity_list = Array containing activity of each variable ;
var_order_list = Initialize variable order according to activity ;
connectivity_list = Initialize to zero for all variables ;
for (i=0; i != number of variables; i++) do
   /* Implicitly, supernode = {var_order_list[0], ..., var_order_list[i]} */
   next_var = var_order_list[i] ;
   correlation_list = find variables connected to least active literal of next_var using
   Variable-Clause database ;
   for all var $\in$ correlation_list do
      connectivity_list[var]++ ; /* Compute correlation */
   end for
   adjust_variable_order(var $\in$ correlation_list) ;
   /* Linear sort is used to update the variable order corresponding to both con-
   nectivity_list as well as activity_list */
   /* Here, {var_order_list[0], ..., var_order_list[i]} is the current variable order*/
end for
return(var_order_list) ;
```

As the next step, those variables are identified that share an edge with the supernode in the set of unresolved clauses. The activity of the literal $\bar{g}$ is greater than the literal g. Hence, only those clauses in which g appears are considered, as they are unsatisfied. These variables are $\{v, e, c, a, b\}$ as shown in Fig.1(d) with highlighted incident edge. Consider the nodes $\{v, e, a, b\}$. They share only one edge with the supernode with weights (correlation) = 1. On the other hand, node

c has two edges incident on the supernode. Therefore, the correlation modeled by both edges should be accounted for. This is computed as the sum of the degrees of correlation between c and the supernode $\{g, u\} = 1{+}1 = 2$. Since this sum is the highest for c, it is selected as the next branching variable. This is shown in Fig. 1(d) as third selection. The supernode set and the *var_ord_list* are correspondingly updated. The above procedure is repeated until all the nodes are ordered and included with in the supernode. The final derived variable order is $\{u, g, c, v, e, a, b, w, d\}$.

Our approach is inspired from the Prim's Minimum Spanning Tree (MST) algorithm [13]. We have named it the **ACCORD** (ACtivity COrrelation ORDering) algorithm. The Pseudo code for the ACCORD algorithm is presented in Algorithm 1. The Variable-Clause database, mentioned in lines 1-2, is implemented using the array of arrays data-structure available within contemporary SAT solvers [2] [14]. The algorithm analyzes the clause-variable database, and for each variable, it computes its correlated variables. Using the activity and correlation measures, the variable ordering is computed. The time complexity of ACCORD can be derived as $O(V \cdot (V \cdot C + V^2))$, where V represents the number of variables and C represents the number of clauses.

3 Experimental Results and Analysis

The ACCORD algorithm has been programmed within the zCHAFF [2] solver using its native data-structures. The algorithm analyzes the constraint-variable relationships of the given CNF-SAT problem and derive a variable order for SAT search. Using this as the initial order, the SAT tool (zCHAFF) performs a search for the solutions. On encountering conflicts, we allow the solver to add conflict-induced clauses and proceed with its book-keeping and (non-chronological) backtracking procedures. In other words, ACCORD provide only an initial static ordering. zCHAFF's VSIDS heuristic updates this order dynamically, when conflict clauses are added. Hence, our approach is not a replacement for VSIDS; it is to be used in conjunction with it. Using this setup, we have conducted experiments over a large set of benchmarks that include: i) Microprocessor verification benchmarks [15]; and ii) some of the hard instances specifically created for the SAT competition (all three categories - industrial, handmade and random). We conducted our experiments on a Linux workstation with a 2.6GHz Pentium-IV processor and 512MB RAM.

Table 1 presents some results that compares the quality of the variable order derived by ACCORD with those of zCHAFF (latest version developed in 2004) and hypergraph partitioning (HGPart) based order [12]. For a fair comparison, zCHAFF is used as the base SAT solver for all experiments - just the variable orders are different. As compared to zCHAFF and HGPart, the variable order generated by ACCORD results in a orders of magnitude speedup in SAT solving. Particularly for more difficult problems, our approach significantly outperforms the other two.

Table 1. Run-time Comparison of ACCORD with zCHAFF and HGPART

		zCHAFF	HGPart + zCHAFF			ACCORD + zCHAFF		
Bench-mark	Vars/ Clauses	Solve (sec)	Var. Time(s)	Solve Time(s)	Total Time(s)	Var. Time(s)	Solve Time(s)	Total Time(s)
3bitadd_31	8432/21K	68.88	3.023	0.55	**3.573**	0.09	4.1	4.19
3bitadd_32	8704/32K	7.43	3.417	0.67	4.087	0.1	0.02	**0.12**
clus-2020-1	1200/4800	>1000	1.421	>1000	–	0.01	666.32	**666.33**
clus-2020-2	1200/4800	688.19	1.81	9.12	**10.93**	0.04	37.17	37.21
clus-1010-3	1200/4919	>1000	1.701	704.66	706.361	0.01	700.85	**700.86**
color-10-3	300/6475	129.57	0.375	19.01	**19.385**	0	25.05	25.05
conn-939	576/6864	135.54	1.196	28.96	30.156	0	0.42	**0.42**
conn-945	596/7157	581.56	1.145	>1000	–	0.01	98.02	**98.03**
mm-1x10	1120/7220	654.16	1.057	>1000	–	0.01	237.48	**237.49**
qwh-35	1597/11K	39.73	1.869	84.59	86.459	0.03	5.58	**5.61**
unif-r4-2	500/2000	>1000	0.447	>1000	–	0	198.47	**198.47**
unif-r4-5	500/2000	884.04	0.491	91.88	**92.371**	0	129.79	129.79
unif-r4-7	500/2000	>1000	0.451	582.53	582.981	0	151.25	**151.25**
unif-r4-9	500/2000	>1000	0.493	711.65	712.143	0	141.95	**141.95**
ferry10	2958/21K	3.54	3.549	0.73	4.279	0.04	0.07	**0.11**
ferry12	4222/32K	238.5	8.189	134.78	142.969	0.09	66.51	**66.6**
icos-stretch	45/352	102.99	0.199	93.84	94.039	0	78.56	**78.56**
marg33-ch	41/272	247.73	0.071	142.53	142.601	0	20.01	**20.01**
marg33add8	41/224	3.44	0.074	1.36	1.434	0	1.29	**1.29**
marg35	61/280	>1000	0.085	>1000	–	0	289.17	**289.17**
mm-2x2-50	60/32000	39.8	3.815	31.37	35.185	0.05	24.9	**24.95**
mm-2x3-66	1698/49K	17.32	4.386	16.48	20.866	0.08	13.55	**13.63**
urqh1c3x3	41/204	284.48	0.078	8.92	**8.998**	0	57.96	57.96
urqh2x4	42/336	302.79	0.077	71.07	71.147	0	26.53	**26.53**
urqh2x5	53/432	>1000	0.117	>1000	–	0	612.96	**612.96**
urqh1c3x4	58/476	>1000	0.143	196.78	**196.923**	0	320.62	320.62
rotmul	5980/35K	174.7	5.888	153.26	**159.148**	0.28	159.53	**159.81**
9dlx-iq1	25K/261K	504.47	68.83	443.79	512.62	15.69	381.09	**396.78**

Table 2 depicts some results for the microprocessor pipeline verification benchmarks. These experiments were conducted using both 2003 and 2004 versions of the zCHAFF SAT solver. For both experiments, the same variable order derived by ACCORD is used. Note that, using the ACCORD's order, zCHAFF-2003 tool is able to improve the performance significantly. zCHAFF-2004 follows upon its earlier versions by implementing the efficient conflict analysis procedures proposed by [3]. As a result for these benchmarks, the order generated by ACCORD gets significantly modified due to the conflict clause resolution [3] and hence, the impact of ACCORD is minimal.

Table 2. Run-time Comparison of ACCORD with zCHAFF'03 and zCHAFF'04

		zCHAFF 2003	ACCORD+ zCHAFF'03			zCHAFF 2004	ACCORD+ zCHAFF'04	
Bench-mark	Vars/ Clauses	Solve (sec)	Var. Time(s)	Solve Time(s)	Total Time(s)	Solve Time(s)	Solve Time(s)	Total Time(s)
4pipe	5K/80K	129	0.56	56.1	56.66	8.02	11.96	12.52
5pipe	9K/195K	196.53	2.36	54.46	56.82	18.34	16.59	18.95
4pipe_k	5K/79K	208.81	0.6	40.85	41.45	8.34	14.53	15.13
5pipe_k	9K/189K	871.79	2.25	388.32	390.57	40.75	31.29	33.54
4pipe_q0	5K/69K	66.71	0.48	73.86	74.34	6.37	8.59	9.07
5pipe_q0	10K/154K	649.9	1.53	214.89	216.42	25.21	26.46	27.99
6pipe	16K/394K	>2000	6.53	827.06	833.59	132.82	144.72	151.25
6pipe_k	15K/408K	105.52	8.76	141.45	150.21	71.49	57.64	66.4
6pipe_q0	17K/315K	104.48	4.88	82.44	87.32	43.37	46.65	51.53
7pipe_q0	26K/536K	>2000	12.45	1523.16	1535.61	284.97	265.3	277.75

4 Conclusions and Future Work

This paper has advocated the need to analyze constraint-variable relationships to derive an ordering of variables to guide SAT diagnosis. To analyze the tightness of the connectivity between variables, we have proposed the degree of correlation as a qualitative and quantitative metric. Our technique models the constraint variable dependencies on a weighted graph and analyzes the graph's topological structure to derive the order. Our approach is fast, robust, scalable and can handle a large set of variables and constraints. The variable order derived by our procedure improves the performance of the solver by one or more orders of magnitude. As part of future work, we are exploring a dynamic variable order update strategy to be employed when conflict clauses are added to the database.

References

1. J. P. M. Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms", *in Portuguese Conf. on Artificial Intelligence*, 1999.
2. M. Moskewicz, C. Madigan, L. Zhao, and S. Malik, "CHAFF: Engineering and Efficient SAT Solver", *in In Proc. Design Automation Conference*, pp. 530–535, June 2001.
3. E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver", *in DATE, pp 142-149*, 2002.
4. J. Marques-Silva and K. A. Sakallah, "GRASP - A New Search Algorithm for Satisfiability", *in ICCAD'96*, pp. 220–227, Nov. 1996.
5. R. Dechter and J. Pearl, "Network-based Heuristics for Constraint-Satisfaction Problems", *Artificial Intelligence*, vol. 34, pp. 1–38, 1987.
6. E. Amir and S. McIlraith, "Solving Satisfiability using Decomposition and the Most Constrained Subproblem", *in LICS workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, 2001.

7. P. Bjesse, J. Kukula, R. Damiano, T. Stanion, and Y. Zhu, “Guiding SAT Diagnosis with Tree Decompositions”, *in Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, 2003.
8. A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, “Partition-based Decision Heuristics for Image Computation using SAT and BDDs”, *in Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design (ICCAD)*, pp. 286–292. IEEE Press, 2001.
9. F. Aloul, I. Markov, and K. Sakallah, “Mince: A static global variable-ordering for sat and bdd”, *in International Workshop on Logic and Synthesis.* University of Michigan, June 2001.
10. J. Huang and A. Darwiche, “A structure-based variable ordering heuristic for SAT”, *in Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1167–1172, August 2003.
11. F. A. Aloul, I. L. Markov, and K. A. Sakallah, “FORCE: A Fast and Easy-To-Implement Variable-Ordering Heuristic”, *in Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pp. 116–119, 2003.
12. V. Durairaj and P. Kalla, “Guiding CNF-SAT Search via Efficient Constraint Partitioning”, *in ICCAD*, pp. 498 – 501, 2004.
13. R. C. Prim, “Shortest connection networks and some generalizations”, *The Bell System Technical Journal*, vol. 36, pp. 1389–1401, 1957.
14. N. Eén and N. Sörensson, “An Extensible SAT Solver”, *in 6th International Conference, SAT*, 2003.
15. M. Velev, “Sat benchmarks for high-level microprocessor validation”, http://www.cs.cmu.edu/ mvelev.

Cost-Effective Hyper-Resolution for Preprocessing CNF Formulas

Roman Gershman and Ofer Strichman

Technion, Haifa, Israel
gershman@cs.technion.ac.il
ofers@ie.technion.ac.il

Abstract. We present an improvement to the HYPRE preprocessing algorithm that was suggested by Bacchus and Winter in SAT 2003 [1]. Given the power of modern SAT solvers, HYPRE is currently one of the only cost-effective preprocessors, at least when combined with some modern SAT solvers and on certain classes of problems. Our algorithm, although it produces less information than HYPRE, is much more efficient. Experiments on a large set of industrial Benchmark sets from previous SAT competitions show that HYPERBINFAST is always faster than HYPRE (sometimes an order of magnitude faster on some of the bigger CNF formulas), and achieves faster total run times, including the SAT solver's time. The experiments also show that HYPERBINFAST is cost-effective when combined with three state-of-the-art SAT solvers.

1 Introduction

Given the power of modern SAT solvers, most CNF preprocessing algorithms are mostly not cost-effective time-wise. Since these solvers are so effective in focusing on the important information in a given CNF, it is particularly challenging to find the right balance between the amount of effort invested in preprocessing and the quality of information gained, in order to positively impact the overall solving time.

The best currently available preprocessor that we are aware of is HYPRE in [1], which is cost-effective when combined with some of the modern SAT solvers, although it fails to be so with very large CNF files. We present a new preprocessing algorithm HYPERBINFAST for CNF formulas which is an improvement of the HYPRE algorithm. Our algorithm makes the preprocessing of CNF formulas cost-effective time-wise by relaxing the optimality constraints presented in the original algorithm. Experiments show that giving up optimality generally improves the overall solving time (preprocessing + SAT). Another advantage of HYPERBINFAST is that it is implemented as an anytime algorithm that can be stopped either according to a predetermined timeout or according to a heuristic function that decides when to stop it by measuring its progress. Due to limit of space we refer the interested reader to the full version of this article [2] for a more detailed description of this heuristic, as well as more experimental results, comparison to previous work and a detailed comparison to HYPRE.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 423–429, 2005.

2 Definitions and Motivation

Definition 1 (Binary Implications graph). *Given a CNF formula φ with a set of binary clauses B, a* Binary Implications Graph *is a directed graph $G(V, E)$ such that $v \in V$ if and only if v is a literal in φ, and $e = (u, v)$ is an edge if and only if B contains a clause $(\overline{u}, v)$.*

A Binary Implications Graph allows us to follow implications through binary clauses. Note that for each binary clause (u, v), both $(\overline{u}, v)$ and $(\overline{v}, u)$ are edges in this graph. Thus, the total number of edges in the initial (before further processing) graph is twice the size of B. This is what we refer to as the *symmetry* of Binary Implications Graphs. Binary Implication Graphs are *Static* and are not related to the standard implication graphs that describe the progress of Unit Propagation (UP - also known as BCP).

Definition 2 (Binary Transitive Closure of a literal). *Given a literal v, a set of literals denoted by $BTC(v)$ is the* Binary Transitive Closure of v *if it contains exactly those literals that are implied by v through the Binary Implications Graph.*

Definition 3 (Failed literal). *A literal v is called a* Failed Literal *if setting its value to $TRUE$ and applying UP causes a conflict.*

Definition 4 (Propagation closure of a literal). *Given a non-failed literal u, a set of literals denoted by $UP(u)$ is the* Propagation closure of u *if it contains exactly those literals that are implied through UP by u in the given CNF (not only the binary clauses).*

It is easy to see that $BTC(v) \subseteq UP(v)$ for every literal v, because $UP(v)$ is not restricted to what can be inferred from binary clauses. Note that $v \in UP(u)$ implies that $u \rightarrow v$ and hence $\overline{v} \rightarrow \overline{u}$, but it is not necessarily the case that $\overline{u} \in UP(\overline{v})$, due to the limitations of UP. For example, in the set of clauses $(\overline{x} \vee y), (\overline{x} \vee z), (\overline{y} \vee \overline{z} \vee w)$, it holds that $x \rightarrow w$ and hence $\overline{w} \rightarrow \overline{x}$, but UP detects only the first direction. It disregards $\overline{w} \rightarrow \overline{x}$ because $\overline{w}$ does not invoke any unit clause. Hence, UP lacks the symmetry of Binary Implications Graphs.

3 The HyperBinFast Algorithm

Algorithm HyperBinFast iterates over all root nodes in the Binary Implications Graph ($roots(v)$ denotes the set of all ancestor roots of v in such a graph). It has two main stages. In the first stage (lines 4 - 5) it iteratively finds equal literals (by detecting SCCs and unifying their vertices to a single 'representing literal'), propagates unit clauses, and simplifies the clauses in the formula. Simplification in this context corresponds to substituting literals by their representative literal in all clauses (not only binary), removing literals that are

```
HyperBinFast
Mark all root vertices as weak;
while there are weak roots, unit clauses, or binary cycles do
    while there are unit clauses or binary cycles do
        Detect all SCCs and collapse each one of them to a single node;
        Propagate all unit clauses and simplify all clauses accordingly;
        For each new binary clause $(u, v)$ mark as weak $roots(\overline{u})$ and $roots(\overline{v})$;
    Choose a weak root node $v$;
    $FailedLiteral$ =FastVisit ($v$);
    Undo assignments caused by BinaryWalk and clear nQueue;
    if $FailedLiteral \neq$ undefined then
        Add unit clause ($\overline{FailedLiteral}$);
    Mark $v$ as strong;
```

```
FastVisit (Literal $v$)
$res \leftarrow$ BinaryWalk ($v$, NULL);
if $res \neq undefined$ then
    return $res$
while !nQueue.empty() do
    Literal $p \leftarrow$ nQueue.pop_front();
    for each n-ary clause $\in$ watched(p) do                    ▷ $n > 2$
        if $clause$ is conflict then
            Literal $fUIP \leftarrow$ FindUIP ($clause$);
            return $fUIP$;
        else if $clause$ is unit then
            Literal $toLit \leftarrow$ undefined literal from $clause$.
            Literal $fromLit \leftarrow$ FindUIP ($clause \setminus \{toLit\}$);
            Add clause ($\overline{fromLit}, toLit$)
            Mark $roots(\overline{toLit}) \cup roots(\overline{fromLit})$ as weak
            $res \leftarrow$ BinaryWalk ($toLit$, ($\overline{fromLit}, toLit$));
            if $res \neq undefined$ then
                return $res$
return $undefined$;
```

```
BinaryWalk (Literal $t$, Antec clause C)
if value($t$)=True then
    return $undefined$;
if value($t$)=False then
    return $\overline{t}$;
$value(t) \leftarrow TRUE$;
$antecedent(var(t)) \leftarrow C$;
Put $t$ on assignment stack;
Put $t$ into nQueue;
for each binary clause $(\overline{t}, u)$ do
    $res \leftarrow$ BinaryWalk ($u$, $(\overline{t}, u)$);
    if $res \neq undefined$ then
        return $res$;
return $undefined$;
```

```
FindUIP (Literal set $S$)
mark all variables in $S$;
$count \leftarrow |S|$;
while $count > 1$ do
    $v \leftarrow$ latest marked variable in the
        assignment stack
    unmark $v$; $count - -$;
    Let $(u, L)$ be antecedent clause of
        $v$, s.t. $var(L) = v$.
    if $var(u)$ not marked then
        mark $var(u)$; $count + +$;
$res \leftarrow$ last marked literal in assign-
    ment stack.
unmark $var(res)$;
return $res$;
```

evaluated to FALSE and removing satisfied clauses. The simplification may result in shortening of some n-ary clauses to binary clauses, which change the Binary Implications Graph. In line 6 we perform a restricted version of what HYPRE does in such cases: while HYPRE marks as *weak* (i.e. nodes that should still be processed) all ancestor nodes, HYPERBINFAST only marks root ancestor nodes. Further, while HYPRE invokes this process every time an n-ary clause is being shortened, HYPERBINFAST only does so for clauses that become binary. The reduced overhead due to these changes is clear. In the second stage (line 8), we invoke FASTVISIT for some weak root node, a procedure that we will describe next. FASTVISIT can change the graph as well, so HYPERBINFAST iterates until convergence.

Computing the Binary Transitive Closure. Before describing FASTVISIT, we concentrate on the auxiliary function BINARYWALK, which FASTVISIT calls several times. The goal of BINARYWALK is to mark all literals that are in BTC(v) or return a failed literal, which can be either v itself or some descendant of v. It also updates a queue, called *nQueue* with those literals in BTC(v) for future processing by FASTVISIT. BINARYWALK performs DFS from a given literal on the Binary Implications Graph. In each recursive-call, if t is already set to FALSE (i.e. $\bar{t}$ is already set to TRUE in the current call to FASTVISIT), it means that there is a path in the binary implication graph from $\bar{t}$ to t, and hence $\bar{t}$ is a failed literal. This is a direct consequence of the following lemma:

Lemma 1. *In a DFS-traversal on a Binary Implications Graph from a literal u that marks all nodes it visits, if when visiting a node t another node $\bar{t}$ is already marked, and this is the* first time *such a 'collision' is detected, then $\bar{t} \xrightarrow{*} t$.*

When BINARYWALK detects such a failed literal it returns $\bar{t}$ all the way out (due to lines 11-12) and back to FASTVISIT and then to HYPERBINFAST.

The other case is when t does not have a value yet. In this case BINARYWALK sets it to TRUE and places it in *nQueue*, which is a queue of literals to be propagated later on by FASTVISIT. It also places t in the (global) assignment stack, and stores for $var(t)$ its antecedent clause (the clause that led to this assignment), both for later use in FINDUIP.

From Binary Transitive Closure to Propagation Closure. We now describe FASTVISIT. Recall that FASTVISIT is invoked for each root node in the Binary Implications Graph. FASTVISIT combines Unit Propagation with Binary Learning based on single assignments, i.e. learning of new clauses by propagating a single decision at a time. It relies on the simple observation that if $u \in UP(v)$ then $v \rightarrow u$. It is too costly to add an edge for every such pair v, u, because this corresponds to at least computing the transitive closure [1]. Since our stated goal is to form a binary graph in which $UP(v) = BTC(v)$ for each root node, it is enough to focus on a vertex u only if $u \in UP(v)$ but $u \notin BTC(v)$. Further, given such a vertex u, although adding the edge $v \rightarrow u$ achieves this goal, we rather find a vertex w, a descendant of v that also implies u, in the spirit of the First Unique-Implication-Point (UIP) scheme. The FINDUIP function called by FastVisit can in fact be seen as a variation of the standard algorithm for finding first UIPs [4]: unlike the standard usage of such a function in analyzing conflicts, here there are no decision levels and the clauses are binary. On the other hand it can receive as input an arbitrary set of assigned literals, and not just a conflict clause.

In line 4 FastVisit starts to process the literals in *nQueue*. For each literal p in this queue, it checks all the n-ary clauses ($n > 2$) watched by p. As usual, each such clause can be of interest if it is either conflicting or unit. If it is conflicting, then FastVisit calls FindUIP, which returns the first UIP causing this conflict. This UIP is a failed literal and is returned to HyperBinFast, which adds its negation as a unit clause in line 11. If the processed clause is a unit clause, the unassigned literal, denoted by *toLit*, is a literal implied by v that is not in $BTC(v)$ (otherwise it would be marked as true in BinaryWalk). In other words, $toLit \in UP(v)$ and $toLit \notin BTC(v)$, which is exactly what we are looking for. At this point we can add a clause $(\overline{v}, toLit)$ but rather we call FindUIP, which returns a first UIP denoted by $fromLit$. The clause $(\overline{fromLit}, toLit)$ is stronger than $(\overline{v}, toLit)$ because the former also adds the information that $\bar{t}olit \rightarrow fromlit$. Note that this is an unusual use of this function, because *clause* is not conflicting. Because the addition of this clause changes the Binary Implications Graph, we need to mark as weak all the ancestor nodes of $fromLit$ and of $\overline{toLit}$, and to continue with BinaryWalk from *toLit*. This in effect continues to compute $BTC(v)$ with the added clause.

4 Experiments, Conclusions and Future Research

Table 1 shows experiments on an Intel 2.5Ghz computer with 1GB memory running Linux. The benchmark set is comprised of 165 industrial instances used in various SAT competitions. The number in brackets for each benchmark set denotes the number of instances. The global timeout for each instance was set to 3000 seconds. The timeout for HyperBinFast was set to 300 seconds, and for Hypre was set to 3000 seconds (Hypre is not implemented as anytime, and only full preprocessing is allowed). The timeout for the SAT solver was dynamically reduced to 3000 minus the time spent during preprocessing. All times in the table include preprocessing time when relevant. We count each failure as 3000 seconds as well. We used our experimental solver HaifaSat which participates in the SAT05 competition and Siege_v1 [3]. The full version of this article includes also a detailed comparison to zChaff 2004. Briefly, zChaff's total run-time is 212,508 sec. (56 fails) and with HyperBinFast it is 179,997 (44 fails).

The table shows that: 1) HyperBinFast helps each of the tested solvers to solve more instances in the given time bound on average 2) When the instance is solvable without HyperBinFast, still HyperBinFast typically reduces the overall run time 3) Whenever HyperBinFast does not help, its overhead in time is relatively small 4) It is very rare that an instance can be solved without HyperBinFast but cannot be solved with HyperBinFast. 5) On average, the total gain in time is about 20-25% relative to the pure configuration. 6) Some-

Table 1. Run-times (in seconds) and failures (denoted by 'F') for various SAT solvers with and without HyperBinFast. Times which are smaller by 10% than in competing configurations with the same SAT solver are bolded. Failures denoted by * are partially caused by bugs in the SAT solver

SAT solver →	HaifaSat						Siege_v1					
Preprocessor →	—		H-B-Fast		Hypre		—		H-B-Fast		Hypre	
	Time	F	Time	F	Time	F	Time	F	Time	F	Time	F
01_rule(20)	19,172	2	**7,379**	0	10,758	1	20,730	4	11,408	1	**5,318**	0
11_rule_2(20)	22,975	6	**7,491**	0	21,247	0	29,303	8	**17,733**	2	20,178	1
22_rule(20)	27,597	8	22,226	5	**16,825**	2	31,839	10	29,044	9	**17,510**	3
bmc2(6)	1,262	0	**81**	0	163	0	3,335	1	**85**	0	156	0
CheckerI-C(4)	**682**	0	902	0	989	0	4,114	0	3,541	0	**2,492**	0
comb(3)	4,131	1	4,171	1	4,056	1	5,679	1	6,027	1	**4,439**	1
f2clk(3)	4,059	1	4,060	1	**3,448**	1	6,105	2	6,063	2	**5,078**	1
fifo8(4)	1,833	0	**554**	0	1,756	0	5,555	1	2,420	0	**1,159**	0
fvp2(22)	1,995	0	2,117	0	3,288	0	1,860	0	2,009	0	2,431	0
hanoi(5)	131	0	285	0	119	0	**357**	0	1,231	0	802	0
hanoi03(4)	**427**	0	533	0	979	0	6,026	2	6,028	2	6,014	2
IBM02(9)	**3,876**	0	5,070	0	11,072	3	10,442	4	**7,881**	0	12,653	3
ip(4)	203	0	**172**	0	365	0	630	0	548	0	**349**	0
pipe03(3)	1,339	0	1,266	0	1,809	0	2,006	0	**1,275**	0	1,822	0
pipe-sat-1-1(10)	**3,310**	0	5,147	0	27,130	10	**2,445**	0	5,249	0	30,029	10
sat02(9)	17,330	4	**14,797**	4	16,669	4	24,182	7	18,843	5	17,662	4
vis-bmc(8)	13,768	3	10,717	2	10,139	2	10,449	2	6,989	1	**5,715**	0
vliw_unsat_2.0(8)	19,425	5	19,862	6	20,421	6	16,983	6	17,891	6	20,375	6
w08(3)	2,681	0	**1,421**	0	2,899	0	4,387	1	**1,711**	0	2,726	0
Total(165)	146,194	30	**108,251**	19	154,132	30	186,426	49	145,978	29	156,910	31

times HyperBinFast is weaker than Hypre (it does not simplify the formula enough), so the SAT solver fails on the corresponding instance but succeeds after applying Hypre. 7) With HaifaSat, Hypre is not cost-effective, neither in the total number of failures or the total run time, while HyperBinFast reduces HaifaSat's failures by 35% and reduces its total solving time by 25%.

Preprocessing Vs. SAT: For the above benchmark, it took HaifaSat 97,909 seconds after HyperBinFast and only 54,567 seconds after Hypre, which indicates that indeed the quality of the CNF generated by Hypre is better, as expected. But these numbers may mislead because, recall, the timeouts for the two preprocessors are different, which, in turn, is because HyperBinFast is an anytime algorithm. In Table 2 we list several benchmarks for which both preprocessors terminated before their respective timeouts, together with the time it took the preprocessor and then HaifaSat to solve them. Therefore, this table shows performance of HyperBinFast comparing to Hypre without taking into consideration the anytime aspect. To the extent that these instances are representative, it can be seen that typically the SAT solving time is longer after

Table 2. Few representative single instances for which both Hypre and HyperBinFast terminated before their respective (different) timeouts. The SAT times refer to HaifaSat's solving time. It can be seen that typically the solving time is longer after HyperBinFast, but together with the SAT solver time it is more cost effective than Hypre

Benchmark:	Hypre	SAT	HyperBinFast	SAT
01_rule.k95.cnf	377	1679	4	1504
11_rule2.k70.cnf	1387	47	71	285
22_rule.k70.cnf	671	251	51	1410
fifo8_400.cnf	164	1226	12	309
7pipe.cnf	651	258	147	416
ip50.cnf	109	82	6	79
w08_14.cnf	1231	5	267	298
Total:	4590	3548	558	4301

HyperBinFast, but together with the SAT solver time it is more cost effective than Hypre.

Conclusions and future research. Our preprocessor HyperBinFast is a compromise between the original Hypre and a pure SAT solver: it tries to benefit from the preprocessing when possible while reducing the overhead when it is not effective. With HyperBinFast, preprocessing is generally cost-effective when combined with modern SAT solvers, as is evident from our experiments with 165 industrial CNF instances from previous SAT competitions. We pointed to two directions for future research: develop more efficient dynamic strategies for determining the amount of time spent for preprocessing, and make preprocessing *decide* on the set of variables from which it begins its traversal of the Binary Implications Graph (and not just choose all the root nodes as we do now). This concept can be generalized to preprocessing in general: while SAT solvers focus on the *semantics* of the formula, that is, they attempt to find the 'important' variables, preprocessors focus on the *syntactical* characteristics of the formula, and are therefore much more sensitive to its size. Hence, attempting to build a *semantic preprocessor* seems like a worthwhile direction to pursue next.

References

1. F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT2003*, volume 2919 of *Lect. Notes in Comp. Sci.*, pages 341–355, 2003.
2. Roman Gershman and Ofer Strichman. Cost-effective hyper-resolution for preprocessing cnf formulas(full version), 2005. www.cs.technion.ac.il/~gershman/papers/sat05_full.pdf.
3. L.Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.
4. J.P.M. Silva and K.A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical Report TR-CSE-292996, Univerisity of Michigen, 1996.

Automated Generation of Simplification Rules for SAT and MAXSAT*

Alexander S. Kulikov

St.Petersburg State University,
Department of Mathematics and Mechanics,
St.Petersburg, Russia
http://logic.pdmi.ras.ru/~kulikov

Abstract. Currently best known upper bounds for many NP-hard problems are obtained by using divide-and-conquer (splitting) algorithms. Roughly speaking, there are two ways of splitting algorithm improvement: a more involved case analysis and an introduction of a new simplification rule. It is clear that case analysis can be executed by computer, so it was considered as a machine task. Recently, several programs for automated case analysis were implemented. However, designing a new simplification rule is usually considered as a human task. In this paper we show that designing simplification rules can also be automated. We present several new (previously unknown) automatically generated simplification rules for the SAT and MAXSAT problems. The new approach allows not only to generate simplification rules, but also to find good splittings. To illustrate our technique we present a new algorithm for $(n, 3)$-MAXSAT that uses both splittings and simplification rules based on our approach and has worst-case running time $O(1.2721^N L)$, where N is the number of variables and L is the length of an input formula. This bound improves the previously known bound $O(1.3248^N L)$ of Bansal and Raman.

1 Introduction

The *splitting method* (i.e., estimating the complexity of divide-and-conquer algorithms by recurrent inequalities) is a powerful tool for proving upper bounds for NP-hard problems. Currently best known upper bounds for many NP-hard problems are obtained by using exactly this method (SAT [7], MAXSAT [3], XSAT [2], MAX-CUT [4], Vertex Cover [6]). Formally, a splitting algorithm splits an input instance of a problem into several simpler instances further simplified by certain simplification rules, such that by finding the solution for all of them one can construct the solution for the initial instance in polynomial time.

* Supported in part by grant No. 1 of the 6th RAS contest-expertise of young scientist projects (1999) and Grant #NSh-2203-2003-1 of the President of Russian Federation for Leading Scientific School Support.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 430–436, 2005.

In general, there are two ways of splitting algorithm improvement: a more involved case analysis and an introduction of a new simplification rule . Recently, several programs for automated case analysis were implemented ([6], [5], [8]). In this paper we present a procedure for generating simplification rules for the SAT and MAXSAT problems. Our approach is based on defining a partial ordering on assignments to variables of a formula and allows not only to generate simplification rules, but also to find good splittings. By using this approach we construct a new algorithm for $(n, 3)$-MAXSAT with worst-case running time $O(1.2721^N L)$. The automated proof of the running time of this algorithm is available at `http://logic.pdmi.ras.ru/~kulikov`.

Motivation. We have already mentioned that one of the two main ways of splitting algorithm improvement is an introduction of a new simplification rule to it. This fact already motivates the need for designing new simplification rules. Also sometimes a new simplification rule does not allow to prove a better upper bound on the running time of a splitting algorithm, but it allows to significantly reduce the case analysis included in this algorithm. In general, a new simplification rule for, say, the SAT problem can improve the running time of a certain SAT solver, not necessarily based on the splitting method.

2 General Setting

2.1 The SAT and MAXSAT Problems

The SAT problem is: given a formula in CNF, check whether this formula is satisfiable, i.e., whether there exists an assignment of Boolean values to all variables of the formula that makes it *True*. For example, the formula $(xy)(\bar{x})$ is obviously satisfiable, while the formula $(xy)(\bar{x})(x\bar{y})$ is not. In the MAXSAT problem one is asked to find a maximal possible number of simultaneously satisfied clauses of a given formula. For example, this number is equal to 3 for the formula $(xy)(\bar{x}\bar{y})(\bar{x}y)(x\bar{y})$.

Both SAT and MAXSAT are very well-known NP-hard problems, that have been attacked by scientists from all over the world for a long time. The currently best known upper bounds for SAT are $O(1.238823^K)$ and $O(1.073997^L)$ [7], and for MAXSAT are $O(1.341294^K)$ [3], $O(1.105729^L)$ [1], where K denotes the number of clauses in the input formula, L denotes the length of the input formula. Note that there are no known upper bounds of the form $O(c^N)$, where $c < 2$ is a constant and N is the number of variables in the input formula, for these problems.

2.2 Simplification Rules

Throughout this paper we consider only simplification rules for SAT and MAXSAT. In case of SAT by an *optimal assignment* we mean an assignment to all variables of a formula that satisfies this formula, in case of MAXSAT we say that an assignment is *optimal* if this assignment satisfies the maximal possible number of clauses of a formula. Note that in case of MAXSAT any CNF formula has an optimal assignment, while in case of SAT only satisfiable formulas have such an assignment.

We say that a simplification rule is applicable to a formula F if it can replace F by another formula F' in polynomial time such that both following conditions hold:

1. the complexity of F' is smaller than the complexity of F;
2. by constructing an optimal assignment for F' one can construct an optimal assignment for F in polynomial time.

2.3 Class of Formulas

Let C be a clause consisting of literals $l_1, l_2, \dots, l_k$. We define a *clause with unfixed length* as a set of clauses that contains all these literals and probably some more literals and denote it by $(l_1 l_2 \dots l_k \dots)$. We call the literals $l_1, l_2, \dots, l_k$ the *basis* of this clause. For example, $(l \dots)$ is the set of all clauses containing the literal l. In the following we use the word "clause" to refer both to clauses in its standard definition and to clauses with unfixed length.

Similarly we define a *class of formulas*. Let $C'_1, \dots, C'_k$ be clauses (some of them may have unfixed lengths). Then a *class of formulas* is the set of formulas represented as $F = \{C_1, \dots, C_m\}$, such that the following conditions hold:

1. $m \geq k$,
2. for $1 \leq i \leq k$, $C'_i \subseteq C_i$ (as sets of literals), if C'_i is a clause with unfixed length, and $C_i = C'_i$ otherwise,
3. for $1 \leq i \leq k$, $k+1 \leq j \leq m$, C_j does not contain any variable from the basis of C'_i.

We denote this set by $C'_1, \dots, C'_k \dots$ and call the clauses $C'_1, \dots, C'_k$ the *basis* of this class of formulas. We define two functions *CorSet* and *CorClause* (for corresponding set and corresponding clause) based on this definition:

$$CorSet(F, \mathcal{F}) = \{C_1, \dots, C_k\} \ ,$$

$$CorClause(C'_i, F, \mathcal{F}) = C_i \ .$$

The notion of class of formulas allows to explain the fact that a formula contains occurrences of certain literals. For example, to show that a formula F contains a $(3,2)$-literal one can write the following:

$$F \in (x \dots)(x \dots)(x \dots)(\bar{x} \dots)(\bar{x} \dots) \dots \ .$$

However, if we want to express the fact that F contains two $(1,1)$-literals that occur together in a clause, we have to write the following:

$$F \in (xy \dots)(\bar{x} \dots)(\bar{y} \dots) \dots \text{ or } F \in (xy \dots)(\bar{x}\bar{y} \dots) \dots \ .$$

For a formula F and a subset S of its variables we define $CorClass(F, S)$ as a class of formulas resulting from F by replacing all its variables that are not in S by "$\ldots$". Clearly, $F \in CorClass(F, S)$. For example,

$$CorClass((xyz)(\bar{x})(\bar{y}zu)(\bar{u}x)(\bar{z}), \{z, u\}) = (z \ldots)(zu \ldots)(\bar{u} \ldots)(\bar{z}) \ldots \ .$$

Note that in most situations we can work with a class of formulas in the same way as with a CNF formula. For example, if we eliminate from the basis of a class of formulas all clauses that contain a literal x and all occurrences of the literal $\bar{x}$ from the other clauses, we obtain the class of formulas resulting from all formulas of the initial class by setting the value of x to $True$. Also it is easy to see that if after assigning a Boolean value to a variable of a class of formulas or applying a (considered in this paper) simplification rule to it, its complexity measure decreases by Δ, then the complexity measure of each formula of this class decreases *at least* by Δ.

For a class of formulas $\mathcal{F}$ and a clause C we define a class of formulas $\mathcal{F} + \{C\}$ as a class resulting from $\mathcal{F}$ by adding the clause C to its basis. Similarly we define a clause with unfixed length $C + \{l\}$, where C is a clause with unfixed length and l is a literal.

We say that a simplification rule is applicable to a class of formulas, if this rule is applicable to every formula of this class. For example, each formula of the class $(x)(x\bar{y} \ldots)(y \ldots) \ldots$ contains a pure literal (i.e., a literal that does not occur negated in a formula).

3 New Simplification Rules

In this section we present several new simplification rules for the SAT and MAXSAT problems.

Usually to prove the correctness of a simplification rule that assigns the value *True* to a literal x one proves that every optimal assignment that contains the literal $\bar{x}$ can be reconstructed into an optimal assignment that contains the literal x. For example, if x is a pure literal of a formula F then any assignment $A \ni \bar{x}$ satisfies not more clauses than the assignment $A \backslash \{\bar{x}\} \cup \{x\}$. This discussion motivates the following definition.

Definition 1. *Let A_1 and A_2 be assignments to all known variables of a class of formulas $\mathcal{F}$. We say that A_1 is stronger than A_2 w.r.t. $\mathcal{F}$ and write $A_1 \succ_{\mathcal{F}} A_2$, if for each formula $F \in \mathcal{F}$ and for each assignment B to all variables of F the following condition holds: if $A_2 \subset B$, then B satisfies not more clauses of F than $B \backslash A_2 \cup A_1$.*

For example, in case of MAXSAT problem if $\mathcal{F}_1 = (x \ldots)(x \ldots) \ldots$, then $\{x\} \succ_{\mathcal{F}_1} \{\bar{x}\}$ (due to the pure literal rule), and if $\mathcal{F}_2 = (xy \ldots)(\bar{x})(x\bar{y} \ldots)(\bar{x})(y \ldots)(x \ldots) \ldots$, then $\{\bar{x}\} \succ_{\mathcal{F}_2} \{x\}$ (due to the almost dominating unit clause rule). The given definition is very easy and natural. However, in general, it is not clear how to check whether one of the given assignments is stronger than the other. Below we show that such a check is possible.

Definition 2. *Let C be a clause and A be an assignment. We define the function $mark(C, A)$ as follows:*

$$mark(C, A) = \begin{cases} \text{“}+\text{”}, & \textit{if } C \textit{ contains at least one literal from } A; \\ \text{“}-\text{”}, & \textit{if } C \textit{ has fixed length and for each literal } x \in C,\ \bar{x} \in A; \\ \text{“?”}, & \textit{otherwise.} \end{cases}$$

For example, $mark((xyz\ldots), \{y\}) = \text{“}+\text{”}$, $mark((x\bar{y}), \{\bar{x}, y\}) = \text{“}-\text{”}$, $mark((x\bar{y}), \{\bar{x}, z\}) = \text{“?”}$, $mark((x\bar{y}\ldots), \{\bar{x}, y\}) = \text{“?”}$. Informally, $mark(C, A)$ just tell us whether clauses of C are satisfied by extensions of A.

Theorem 1. *Let A_1, A_2 be assignments to all variables of a class of formulas $\mathcal{F}$ and let*

$$p_1 = |\{C \in \mathcal{F} \mid mark(C, A_1) = \text{“}+\text{”}\}| \ ,$$

$$p_2 = |\{C \in \mathcal{F} \mid mark(C, A_2) = \text{“}+\text{”}\}| \ ,$$

$$q = |\{C \in \mathcal{F} \mid mark(C, A_2) = \text{“?”}, mark(C, A_1) \neq \text{“?”}\}| \ .$$

Then, $A_1 \succ_{\mathcal{F}} A_2$ iff $p_1 \geq p_2 + q$.

Let us show how this theorem allows to generate simplification rules. We start with giving several examples. Suppose $F \in \mathcal{F} = (xy)(x\bar{y})(\bar{x}y)(\bar{x}\bar{y}\ldots)\ldots$. In such a case we can replace F by $F[\bar{x}, \bar{y}]$, since $\{\bar{x}, \bar{y}\} \succ_{\mathcal{F}} \{x, y\}, \{\bar{x}, y\}, \{x, \bar{y}\}$. The assignment $\{\bar{x}, \bar{y}\}$ satisfies *at least* three clauses from $CorSet(F, \mathcal{F})$, while any other assignment to variables x and y satisfies *at most* three of these clauses.

Consider a slightly more complicated example. Let $\mathcal{F} = (xy\ldots)(x\ldots)(\bar{x}\bar{y}\ldots)(\bar{y})\ldots$, then $\{\bar{x}, y\} \succ_{\mathcal{F}} \{\bar{x}, \bar{y}\}$ and $\{x, \bar{y}\} \succ_{\mathcal{F}} \{x, y\}$. It means that for any formula $F \in \mathcal{F}$, if there exists an optimal assignment for F, then there exists an optimal assignment B, such that either $\{\bar{x}, y\} \subset B$ or $\{x, \bar{y}\} \subset B$. Thus, we can replace F by $F[y = \bar{x}]$.

In both given examples we construct a *majority set* of assignments to all variables of a class of formulas. Below we formally define this notion.

Definition 3. *Let $\mathcal{F}$ be a class of formulas, M_0 be the set of all possible assignments to all variables of $\mathcal{F}$. We say that $M \subset M_0$ is a* majority set *for $\mathcal{F}$ and write $M = MajorSet(\mathcal{F})$, if for any assignment $A_0 \in M_0$ there exists an assignment $A \in M$, such that $A \succ_{\mathcal{F}} A_0$.*

It is easy to see that a majority set is not unique and that, in particular, M_0 (from the definition) is a majority set for $\mathcal{F}$. However, we are interested in majority sets of small size. We use a greedy algorithm for construction of such majority sets, its code is given in Fig. 1.

Now we are ready to present the new simplification rule that unifies many known simplification rules that modify an input formula by assigning a value to a literal. It is illustrated in Fig. 2. It is easy to see that the running time

```
Procedure MSC

Input: A class of formulas $\mathcal{F}$.
Output: A majority set for $\mathcal{F}$.
Method.

1. let $M_0 = 0$
2. let $M$ be the set of all possible assignments to all variables of $\mathcal{F}$
3. in case of SAT: remove from $M$ all assignments $A$, such that $mark(C, A) = \text{“} - \text{”}$
   for some clause $C \in \mathcal{F}$
4. while $M \neq 0$
   (a) let $A_0$ be an assignment of $M$ such that $|\{A \in M | A_0 \succ_{\mathcal{F}} A\}|$ is maximal
   (b) $M_0 = M_0 \cup \{A_0\}$
   (c) $M = M \backslash \{A \in M | A_0 \succ_{\mathcal{F}} A\}$
5. return $M_0$
```

Fig. 1. The greedy algorithm for majority set construction

```
Procedure GSR(integer $c$)

Input: A CNF formula $F$.
Output: A simplified CNF formula $F$.
Method.
for each subset $S$ of variables of $F$, such that $|S| \leq c$

1. let $\mathcal{F} = CorClass(F, S)$
2. let $M = MajorSet(\mathcal{F})$
3. if there is a literal $x$, such that it occurs in every assignment from $M$, then return
   $F[x]$
4. if there are literals $x$ and $y$, such that for any assignment $A \in M$, either $\{x, y\} \subset A$
   or $\{\bar{x}, \bar{y}\} \subset A$, then return $F[x = y]$
```

Fig. 2. The general simplification rule

of this rule is $O(L)$, the correctness is trivial. We call this rule an *automated* procedure for generating simplification rules, as one can add this procedure with any constant parameter into an algorithm. Clearly, if $GSR(c_1)$ can simplify a formula, then $GSR(c_2)$ can simplify this formula too, for $c_1 < c_2$.

Below we give several new simplification rules, that are particular cases of GSR.

New simplification rules for MAXSAT:

1. if $F \in \mathcal{F} = (x)(\bar{x}y \ldots)(\bar{x}\bar{y} \ldots)(\bar{x}\bar{y} \ldots)(\bar{y} \ldots)(\bar{y} \ldots) \ldots$, then replace F by $F[x]$ (since $\{\{x, y\}, \{x, \bar{y}\}\} = MajorSet(\mathcal{F})$)
2. if $F \in \mathcal{F} = (xyz \ldots)(\bar{x}z \ldots)(\bar{x}z \ldots)(y \ldots)(\bar{y}\bar{z} \ldots)(z \ldots) \ldots$, then replace F by $F[x = y]$ (since $\{\{\bar{x}, y, \bar{z}\}, \{x, \bar{y}, z\}, \{x, y, z\}\} = MajorSet(\mathcal{F})$)

3. if $F \in \mathcal{F} = (x)(\bar{x}yz\ldots)(\bar{x}\bar{t}\ldots)(\bar{x}\bar{t}\ldots)(yt\ldots)(\bar{y}\bar{t}\ldots)(\bar{y}\ldots)(z\bar{t}\ldots)(\bar{z}\ldots)(\bar{z}\ldots)\ldots$, then replace F by $F[x]$ (since $\{\{x, \bar{y}, \bar{t}\}, \{x, y, \bar{t}\}, \{x, \bar{y}, t\}\} = MajorSet(\mathcal{F})$)

New simplification rules for SAT:

1. if $F \in \mathcal{F} = (xy)(\bar{x}\ldots)(\bar{y}\ldots)(\bar{x}y\ldots)(\bar{y}\ldots)\ldots$, then replace F by $F[x = \bar{y}]$ (since $\{\{x, \bar{y}\}, \{\bar{x}, y\}\} = MajorSet(\mathcal{F})$)
2. if $F \in \mathcal{F} = (xy\ldots)(xz\ldots)(\bar{x}\bar{z}\ldots)(\bar{x}\bar{z}\ldots)(\bar{x}\bar{z}\ldots)(y\ldots)(y\ldots)(\bar{y}z\ldots)(\bar{y}\ldots)(\bar{z}\ldots)\ldots$, then replace F by $F[x = \bar{z}]$ (since $\{\{x, \bar{y}, \bar{z}\}, \{x, y, \bar{z}\}, \{x, \bar{y}, \bar{z}\}\} = MajorSet(\mathcal{F})$)

Actually our technique allows to find not only simplifications, but also good splittings. The main idea is that for any formula $F \in \mathcal{F}$ the splitting $F[A_1]$, $F[A_2], \ldots, F[A_k]$ is correct, where $\{A_1, A_2, \ldots, A_k\} = MajorSet(\mathcal{F})$.

Acknowledgements

The author would like to express a great gratitude to his supervisor Edward A. Hirsch for valuable comments.

References

1. N. Bansal and V. Raman. Upper bounds for MaxSat: Further improved. In *Proceedings of ISAAC'99*, pages 247–258, 1999.
2. J. M. Byskov, B. A. Madsen, and B. Skjernaa. New algorithms for exact satisfiability. Technical Report RS-03-30, BRICS, 2003.
3. J. Chen and I. Kanj. Improved exact algorithms for MAX-SAT. In *Proceedings of the 5th LATIN*, volume 2286 of *LNCS*, pages 341–355, 2002.
4. S. S. Fedin and A. S. Kulikov. A $2^{|E|/4}$-time algorithm for MAX-CUT. *Zapiski nauchnykh seminarov POMI*, 293:129–138, 2002.
5. S. S. Fedin and A. S. Kulikov. Automated proofs of upper bounds on the running time of splitting algorithms. *Zapiski nauchnykh seminarov POMI*, 316:111–128, 2004.
6. J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Automated generation of search tree algorithms for hard graph modification problems. *Algorithmica*, 39(4):321–347, 2004.
7. E. A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.
8. S. I. Nikolenko and A. V. Sirotkin. Worst-case upper bounds for sat: automated proof. In *Proceedings of the 8th ESSLLI Student Session*, pages 225–232, 2003.

Speedup Techniques Utilized in Modern SAT Solvers –An Analysis in the MIRA Environment–

Matthew D.T. Lewis, Tobias Schubert, and Bernd W. Becker

Chair of Computer Architecture,
Institute of Computer Science,
Albert-Ludwigs-University of Freiburg, Germany
{lewis,schubert,becker}@informatik.uni-freiburg.de

Abstract. This paper describes and compares features and techniques modern SAT solvers utilize to maximize performance. Here we focus on: Implication Queue Sorting (IQS) combined with Early Conflict Detection Based BCP (ECDB); and a modified decision heuristic based on the combination of Variable State Independent Decaying Sum (VSIDS), Berkmin, and Siege's Variable Move to Front (VMTF). These features were implemented and compared within the framework of the MIRA SAT solver. The efficient implementation and analysis of these features are presented and the speedup and robustness each feature provides is demonstrated. Finally, with everything enabled (ECDB with IQS and advanced decision heuristics), MIRA was able to consistently outperform zChaff and even Forklift on the benchmarks provided, solving 37 out of 111 industrial benchmarks compared to zChaff's 21 and Forklift's 28.

1 Introduction

SAT solvers have advanced considerably since the days of the original Davis-Putnam (DP) algorithm [5,6]. While the underlying algorithm has not changed, improvements in its implementation have made it practical. SAT solvers are now commonly used in industrial fields such as EDA. Such features as Early Conflict Detection BCP (ECDB) [4], Conflict Analysis [7], watched literals [1,21], advanced decision heuristics, and others are what have made SAT solvers practical. Now, while these techniques are powerful, they must all be used together in unison, increasing the complexity of modern SAT solvers. This paper tries to clarify and explain many of the features found in modern SAT solvers while including features only found in our MIRA solver such as the novel Implication Queue Sorting (IQS).

The next section focuses on IQS followed by a section on ECDB. Section 4 will discuss the novel hybrid decision heuristic used in MIRA and how random restarts can be used. Section 5 covers conflict analysis and conflict clause database management. After presenting the parts of a solver, a comparison of MIRA and other solvers on Industrial and Handmade benchmarks are given. Lastly, this paper assumes the reader has a good understanding of the inner workings of a SAT solver and it therefore will not present an overview of current SAT solvers. An overview can be found in [16].

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 437 – 443, 2005.

2 Implication Queue Sorting and the Start of ECDB

Before discussing IQS and ECDB directly, it is important to understand some of the reasons for using it. The idea behind ECDB is to find good conflicts while reducing the work the BCP procedure must do in order to find them. However, in most solvers the BCP procedure checks many clauses that will rarely or never lead to a conflict. This is inefficient and slows down the solver.

Normally, the BCP starts by checking all clauses that contain the current decision variable as a watched literal. As it checks these clauses, it will normally find many implications that it will add to the implication queue. When it is done checking the current decision variable it will move on and check the first implication in the implication queue. The problem with this method is that the BCP procedure will check many implications that are unrelated to the current search space. So, to try and reduce the number of implications the BCP must process, MIRA introduces Implication Queue Sorting (IQS). Simply put, the BCP procedure in MIRA uses a heuristic to select which implications to check first from the implication queue. This is in contrast to other solvers which process the implication queue in chronological order. This results in a significant speedup.

However, to realize this speedup, IQS must be implemented efficiently as it is an operation that is performed frequently. Implications are added and removed from the queue often, the queue can grow quite large (100's of implications), and is reset after every conflict. In MIRA, a reverse VSIDS algorithm is used when selecting the implication to check next. Instead of taking the literal with highest score, MIRA selects the literal whose complement has the highest score (using the same VSIDS counters as the decision heuristic). The idea is that as soon as the BCP starts, the solver is now trying to find a conflict, not solve the problem.

Queue Number	Value		Array Location			
			0	1	…	m
4	Score >3	→	Counter	Impl. 0	…	Impl. n
3	Score = 3	→	Counter	Impl. 0	…	Impl. n
2	Score = 2	→	Counter	Impl. 0	…	
1	Score = 1	→	Counter	Impl. 0	…	Impl. n
0	Score = 0	→	Counter	Impl. 0	…	Impl. n

Fig. 1. MIRA's Implication Queue

MIRA uses five separate queues to efficiently implement this heuristic as shown above. Implications are added to each queue depending on their score. If an implication has a score of 2, it is inserted into queue 2. The counter variable is then incremented to keep track of the number of implications in that queue. This is true for scores 0 to 3. If the score is higher, it is put into queue 4. Using this queue system, it is easy to insert (only one comparison is needed), reset (reset the length of each queue to 0), or select. Selecting from the queues is simple, as the solver takes a literal from the highest nonempty queue. For queues 0 to 3 it can pick any literal, however, if it chooses queue 4, the solver must search the entire queue for the best choice.

The aforementioned queue system was developed after observing MIRA on multiple benchmarks. About 90% of the implications added to MIRA's implication queue have a score of 0. This is significant, because it means that 90% of the implications checked by the BCP procedure have not created or participated in a conflict recently. Note, in MIRA the VSIDS counters are even incremented when a variable is looked at when creating the implication graph. So, if the implication queue contains 100 literals, 90 of them will probably have nothing to do with the current search space. The remaining 10 nonzero score literals are more interesting, and the solver searches these implications first. Queues 1-3 normally take care of approximately 8-9% of the implications, leaving only 0.5-2% of the literals, which are put into the 4th queue. The 4th queue must be searched in a linear manner. Also, when using full ECDB, there is no need for an implication queue for literals with a score of zero as these implications are already in the decision stack and can be checked in chronological order.

An implication queue can contain multiple conflicts, some more helpful than others. Certain solvers [8] search for multiple conflicts, recording one or multiple conflict clauses. Since MIRA checks implications that are more relevant to the current search space first (relevant as their scores are higher), it should in theory find conflicts that are more related to the current search space. Also, by choosing the implication this way, we are trying to unsatisfy clauses. This helps the BCP procedure find more unit clauses, causing the implication queue to grow faster, aiding ECDB while also giving the IQS algorithm more literals to choose from.

The overhead required to implement IQS is quite low. Using the data structure above, literals can be added in constant time. After a conflict, the queues can also be reset in constant time by just resetting the length counter. Approximately 98% of the implications can be added, searched for, and removed in constant time. It is only with the small last group that a linear search must be performed. During testing, this method only searched 1.46 implications on average to decide which literal should be checked by the BCP. This equates to approximately zero overhead, and all functions relating to IQS require only 1% of the computation time (Gprof) which is an insignificant increase when compared to a solver without IQS, as they still need to add and remove literals from an implication queue.

3 MIRA and Early Conflict Detection BCP

To try and reduce the work of the BCP, MIRA introduced ECDB. Certain parts of ECDB have been implemented in other solvers, but MIRA was the first with a comprehensive implementation. MIRA still uses many of the ideas that zChaff introduced relating to fast BCP and watched literal lists. Building on the ideas from zChaff, MIRA introduced a smarter BCP that detects conflicts sooner by utilizing more of the information available to it when it evaluates a clause, thereby reducing the total number of evaluated clauses. The first part of ECDB was IQS and was described above. The following section will give a quick overview of ECDB.

3.1 Full ECDB

The ECDB procedure is still similar to zChaff when it traverses the watched literal list of the current decision variable. However, as discussed above, MIRA does not go chronologically through the implication queue like zChaff does, using IQS instead. Full ECDB also tries to utilize all the information in the implication queue. It does this by using the implication queue during the entire process when evaluating a clause: choosing a new second watched literal; finding conflicts; or new implications. By not ignoring the implication queue during the evaluation of each clause as zChaff does, the BCP procedure can make better decisions when changing watched literals. MIRA will not replace a watched literal with a literal that is falsely defined in the implication queue. This then avoids a re-evaluation of the clause in the near future. Also, in the same manner, the implication queue can aid in the evaluation of a clause detecting unit clauses earlier. This, with the help of IQS, increases the growth rate of the implication queue further aiding the ECDB procedure. This all leads to Full ECDB's ability to detect conflicts sooner. Details of the implementation and analysis of this ECDB procedure without IQS is discussed in [4].

3.2 Other BCP Features

MIRA's ECDB procedure contains other optimizations. It has a regular ECDB procedure for handling all clauses except single literal clauses. MIRA assigns the literals in these type of clauses to decision level 0 and then restarts. This simplifies the regular ECDB procedure as it can assume every clause has at least 2 literals.

MIRA also contains a second BCP for handling binary clauses that are contained in the original problem. This is done because a Binary BCP procedure consists only of arrays of implications that are forced by a certain decision. These arrays can be processed very fast and efficiently. The Binary BCP is also run first before the regular BCP in order to populate the decision stack with short forcing clauses. This should in theory aid the creation of shorter conflict clauses. This type of optimization is also done in other solvers (e.g. jerusat [12], zChaff 2004 [8]).

4 Decision Heuristic and Restarts

The next area where MIRA differs from zChaff is related to decision making. MIRA contains 2 decision strategies, but 3 different heuristics. This is normal as many solvers use multiple decision strategies to improve performance [8,14,15,16]. MIRA's first decision strategy is a combination of VSIDS [1] and Berkmin [2]. When VSIDS returns a decision with a score of 0, the Berkmin algorithm is used. This is opposite to zChaff as we observed that a large number of decisions made by VSIDS have a score of 0, especially if the counters are reduced frequently. Decisions made with variables with a score of zero are not that helpful. The second decision strategy MIRA uses is VMTF [3]. In MIRA, to further increase the robustness of this strategy, literals that are part of the conflict analysis but do not appear in a conflict clause are moved up in the list by a constant amount, and not to the top like conflict clause literals. MIRA switches decision strategies every few restarts, normally making thousands of decisions before switching decision strategies.

5 Conflict Analysis and Clause Deletion

The conflict analysis algorithm that is used in MIRA is implemented differently than that of zChaff's in order to handle MIRA's implication queue, but the results for a give implication graph are the same. Both algorithms generate conflict clauses based on the first UIP. The first UIP conflict analysis is well explained in [18]. For a more in-depth look at MIRA's conflict resolution procedure refer to [4].

While a conflict resolution procedure is an essential part of any modern SAT solver, if left unchecked, the clauses it produces will eventually overwhelm the BCP procedure. MIRA uses a clause aging method which is similar to Berkmin.

6 Results and Comparisons

MIRA_V2 was written in C++ and compared against zChaff version 2004.11.15. This comparison was run on an AMD Opteron 2GHz with 4GB of RAM. Benchmarks were selected from the SAT 2004 competition [17]. As there is no source or executable available for Forklift [20], its results were taken directly from the SAT 2004 competition. The machine used there was an Intel 2.4GHz Pentium 4.

First, Table 1 starts with the more interesting Industrial benchmarks followed by Handmade benchmarks. In the Industrial category MIRA significantly beats zChaff and is more robust and consistent when relating to shuffling. Forklift's performance is slightly better, matching MIRA's performance in 5 categories with the same number of problems solved. On problem sets where Forklift tied MIRA on problems solved it was normally slightly faster time wise. However, when it comes to total Industrial problems solved, Forklift is slower. Note again that Forklift was run on a different machine. However, our machine and the Intel machine should produce similar results. As for Handmade benchmarks, MIRA does better than either of the other two solvers, losing to Forklift only once.

Table 2 then shows a feature comparison of MIRA. The benchmarks used for this table are the Industrial benchmarks from Table 1. The results of this table were made by disabling one feature from MIRA at a time to see the impact each feature has on performance. The first column is the reference MIRA version with everything enabled solving 37 problems. The next column is with only random restarts disabled and everything else enabled. As can be seen, random restarts help significantly. The No IQS column is MIRA without IQS, and only without it. Next, with the Binary BCP disabled all clauses are processed with the normal BCP procedure slowing things down. The last two columns compare the two decision strategies. When only VSIDS or VMTF is enabled, there are no second decision strategies or implication queue strategies, only the default VSIDS or VMTF is used. First, MIRA's performance using only VSIDS is very similar to that of zChaff with respect to the number of problems solved. MIRA's performance with VMTF is very similar to that of Forklift. The point of this table is to show that without all the features enabled, it is very hard to solve over 30 instances. To solve the hard instances, all features must be enabled.

7 Conclusion

This paper presented advanced techniques used in current SAT solvers. It showcased novel ideas developed that allowed MIRA to achieve the performance and robustness that it does. The ideas presented here when used in unison can allow a SAT solver to achieve significant speedup. This was clearly shown in Section 7 when comparing

MIRA to other modern solvers. Lastly, this paper was written to try and lower the barrier to entry for new solvers by explaining what current solvers do, hopefully allowing more solvers to be more competitive, and thereby advancing the entire field.

References

1. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver", Proceedings of the 38th DAC, July 2001.
2. 2.. E. Goldberg and Y. Novikov, "BerkMin: a Fast and Robust Sat-Solver", DATE 2002, Paris, France, March 2002, pp. 142-149.
3. R. Lawrence, “Efficient Algorithms for Clause-Learning SAT Solvers”, Master’s thesis, Simon Fraser University, February 2004.
4. M. Lewis, T. Schubert, and B. Becker, “Early Conflict Detection Based BCP for SAT Solving”, SAT 2004, May 2004.
5. M. Davis, and H. Putnam, “A Computing Procedure for Quantification Theory”, Journal of the ACM, Vol. 7, Number 3, pp. 201-215, July 1960.
6. M. Davis, G. Logemann, and D. Loveland, “A Machine Program for Theorem-Proving”, Communications of the ACM, vol. 5, pp 394-397, 1962.
7. J. P. Marques-Silva,K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability", IEEE Transactions on Computers, Vol. 48, pp. 506-521, 1999.
8. Z. Fu, Y. Mahajan, S. Malik, “New Features of the SAT’04 Version of zChaff”, SAT 2004 Competition: Solver Descriptions, May 2004.
9. A. Biere, “The Evolution from Limmat to Nanosat”, Technical Report 444, Dept. of Computer Science, ETH Zürich, 2004.
10. A. Biere, Limmat Solver, http://www.inf.ethz.ch/personal/biere/projects/limmat/
11. A. Van Gelder, “Generalizations of Watched Literals for Backtracking Search”, 2001.
12. A. Nadel, “The Jerusat SAT Solver” Master’s thesis, Hebrew University of Jerusalem, 2002.
13. I. Lynce, J. and Marques-Silva, "Efficient Data Structures for Fast SAT Solvers", 2001.
14. J. Alfredsson, “The SAT Solver Oepir”, SAT 2004 Competition: Solver Descriptions, 2004.
15. Niklas Eén, Satzoo, http://www.cs.chalmers.se/~een/Satzoo/
16. N. Eén, and N. Sörensson, "An Extensible SAT-solver", SAT 2003, May 5-8 2003.
17. SAT2004, www.satlive.org
18. L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver", ICCAD 2001.
19. H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman, “Dynamic Restart Policies”. The Eighteenth National Conference on Artificial Intelligence, 2002.
20. E.Goldberg, Y.Novikov, http://eigold.tripod.com/BerkMin.html
21. L. Zhang, and S. Malik, "Cache Performance of SAT Solvers: A Case Study for Efficient Implementation of Algorithms", SAT 2003, May 5-8 2003.

Table 1. MIRA vs zChaff

Industrial	# of Instances	zChaff 15.11.2004		MIRA_V2		ForkLift	
		# Solved	Time	# Solved	Time	# Solved	Time
Hoonsang /vis-bmc	8	**3**	**3534,62**	**3**	3614,92	**3**	**3295,98**
./shuffling-1	8	**3**	**3461,61**	**3**	3563,82	**3**	**3228,91**
./shuffling-2	8	2	3861,59	**3**	**3612,49**	2	3764,52
Vangelder /cnf-color	12	1	6600,50	**4**	**5255,62**	3	5406,83
./shuffling-1	12	1	6600,08	**4**	**5055,02**	2	6005,58
./shuffling-2	12	1	6600,21	**4**	**5324,48**	3	5770,96
Velev /pipe-sat-1-1	12	3	4596,59	**6**	**3371,43**	3	4368,47
./shuffling-1	12	**4**	4570,03	**4**	**4533,03**	**4**	**4460,95**
./shuffling-2	12	3	5098,19	**4**	**4629,25**	**4**	**4430,39**
Simon03/sat02bis-shuffling-1	10	0	6000,00	**1**	**5489,85**	0	6000,00
Schuppan03/l2s-shuffling-2	11	0	6600,00	**1**	**6367,17**	**1**	6398,40
Subtotal	111	21	57523,42	**37**	**50817,08**	28	53730,99
Handmade							
Anton/l3k3	9	**9**	202,36	**9**	**144,28**	**9**	160,57
Anton/l4k4	9	3	**3838,81**	**4**	3844,03	2	4662,42
Connamacher /connm-d0.00	9	**2**	**4687,35**	**2**	4712,75	1	4817,25
Connamacher /connm-d0.04	9	**6**	1859,29	**6**	**1812,55**	**6**	1851,12
Gom16/bqwh	8	0	4800,00	**1**	**4408,05**	0	4800,00
Kexu/frb-0.8	12	0	7200,00	**2**	**6377,02**	**3**	**6359,62**
Subtotal	56	20	22587,81	**24**	**21298,68**	21	22650,98
Total	167	41	80111,23	**61**	**72115,76**	49	76381,97

Table 2. MIRA Feature Comparison

	MIRA_V2	No Restarts	No IQS	No BinBCP	Only VSIDS	Only VMTF
Problems Solved	37	28	31	30	22	29

FPGA Logic Synthesis Using Quantified Boolean Satisfiability

Andrew Ling[1], Deshanand P. Singh[2], and Stephen D. Brown[2]

[1] University of Toronto, Toronto ON, Canada
aling@eecg.toronto.edu
[2] Altera Corporation, Toronto Technology Centre, Toronto ON, Canada
{dsingh, sbrown}@altera.com

Abstract. This paper describes a novel Field Programmable Gate Array (FPGA) logic synthesis technique which determines if a logic function can be implemented in a given programmable circuit and describes how this problem can be formalized and solved using Quantified Boolean Satisfiability. This technique is general enough to be applied to any type of logic function and programmable circuit; thus, it has many applications to FPGAs. The applications demonstrated in this paper include FPGA technology mapping and resynthesis where their results show significant FPGA performance improvements.

1 Introduction

FPGAs are integrated circuits characterized by two distinct features: programmable logic blocks and programmable interconnect structures. An example of a programmable logic block (PLB) is shown in Fig. 1. The logic block is composed of a 4-input lookup table (4-LUT) that is capable of implementing any arbitrary Boolean function of 4 variables.

In general, many modern PLBs are based on the k-input lookup table (k-LUT) which contains 2^k SRAM bits. Although the k-input LUT is very flexible, it is usually beneficial to add dedicated non-programmable logic to the PLB

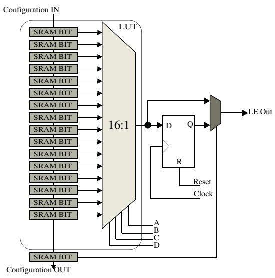


Fig. 1. A basic FPGA PLB

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 444–450, 2005.

such as adders and XOR/AND-gates [1]. These features increase the number of functions that can be implemented by a PLB without the power, speed, and area costs associated with programmable logic. However, because this reduces the flexibility of the PLB, optimal mapping of functions to these non-programmable components is difficult.

The cost of implementing a circuit in an FPGA is directly proportional to the number of PLBs required to implement the functionality of the circuit. FPGAs are sold in a number of pre-fabricated sizes. Decreasing the number of PLBs may allow a circuit to be realized in a smaller FPGA.

The technology mapping step of the FPGA CAD flow takes a gate-level representation of a circuit and produces a netlist of PLBs which implements the same functionality as the gate-level description. Technology mapping has a significant impact on the number of PLBs required to realize a particular circuit. In this paper we will show how methods based on Quantified Boolean Satisfiability solve this problem.

The goal technology mapping step is to reduce area, delay, or a combination thereof in the PLB network that is produced. The process of technology mapping is often treated as a covering problem. This is illustrated in Fig. 2 which shows a possible covering using 4-LUTs. The process illustrated in Fig. 2 is much more difficult for general PLBs since not every cone can be implemented in a PLB. Thus, a legality check must be done to discard illegal cones found in the covering. We pose this problem as a Quantified Boolean Formula (QBF) which can be solved using Quantified Boolean Satisfiability (QSAT).

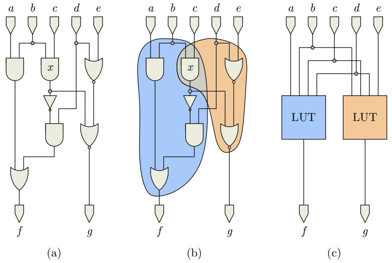


Fig. 2. (a) The initial netlist. (b) Possible covering. (c) The mapping produced

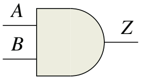
2 Formalizing FPGA Technology Mapping

Assuming that a programmable circuit can be represented as a Boolean function $G_{circuit} = G(x_1..x_n, L_1..L_m, z_1..z_o)$ where $x_i, L_j, z_k, G_{circuit}$ represent the input signals, configuration bits, intermediate circuit signals, and output function of the circuit respectively, the problem of function mapping into programmable logic can represented formally as a QBF as follows.

$$\exists L_1 ... L_m \forall x_1 ... x_n \exists z_1 ... z_o (G_{circuit} \equiv F_{function}) \tag{1}$$

A satisfying assignment to Equ. 1 implies that $F_{function}$ can be realized in the programmable circuit. A detailed explanation of this can be found in [7, Ch.2].

In order to derive Equ. 1, the proposition $(G_{circuit} \equiv F_{function})$ must be represented as a CNF Boolean formula. This can be done using a well known derivation technique that converts logic circuits into a *characteristic function* in CNF [6]. This CNF formula describes all valid inputs, output, configuration bits, and internal signal vectors for the configurable circuit. For example, consider Fig. 3. It shows the basic CNF formula for an AND gate.



$$F_1(A, B, Z) = (A + \overline{Z}) \cdot (B + \overline{Z}) \cdot (\overline{A} + \overline{B} + Z) \tag{2}$$

Fig. 3. AND gate CNF formulae

Equ. 2 will be satisfied if and only if the variables representing each input, output, and internal wires hold a feasible logic representation. For the single AND gate, this includes all input and output combinations that are consistent with the AND function (e.g. $(A = 0, B = X, Z = 0)$, $(A = X, B = 0, Z = 0)$ or $(A = 1, B = 1, Z = 1)$.

The previous conversion technique for the AND structure can be extended to much larger circuits such as PLBs. This creates a characteristic function, Ψ, dependent on variables $x_1, ..., x_n$, $L_1, ..., L_m$, $z_1, ..., z_o$, and G which represent the inputs, programmable bits, intermediate wires, and output of the circuit respectively. The proposition $(G_{circuit} \equiv F_{function})$ can be formed using Ψ by substituting all instances of the output variable G in Ψ with the expression representing $F_{function}$. As an optional step, the quantifiers in Equ. 1 can be removed to form a SAT problem [7, Ch.2]. This, however, is only practical for a small number of input variables $(x_1 ... x_n)$ but often is faster in solving the original QBF.

3 Quantified SAT FPGA Applications

3.1 Application 1: General Technology Mapping

Our first application of QSAT is technology mapping. We developed an algorithm called SATMAP[1] that is based upon IMap [8, Ch.4], an iterative k-LUT

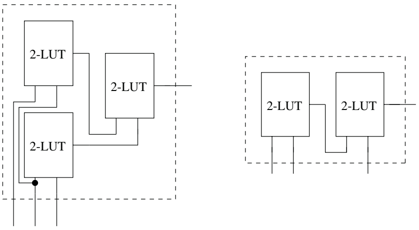
[1] The quantifiers in the QBF were removed so SAT could be used to solve the cone problem. This turned out faster than using naive QSAT.

technology mapping algorithm. For a detailed description of IMap please refer to [8, Ch.4]. The key difference between SATMAP and IMap is how SATMAP generates cones. IMap only dealt with k-LUTs, hence all cones with k-inputs or less (k-feasible cone) were legal; however, SATMAP is a PLB technology mapper, thus not all k-feasible cones are legal. After a set of all k-feasible cones is generated, in contrast to IMap, SATMAP adds a QBF based legality check to every cone where cones that cannot fit into the PLB structure are discarded.

3.2 Application 2: Area-Driven Resynthesis for k-LUTs

When resynthesizing a LUT network for area, one must attempt to reduce the number of LUTs in the network yet maintain the original functionality. The more LUTs that can be removed, the farther the original circuit was from the optimal mapping. Removing LUTs can be achieved by resynthesizing smaller subcircuits and applying this in a sliding window fashion over the larger circuit. These subcircuits form a cone, therefore by resynthesizing several cones, this will reduce the LUT count of the overall LUT network. This is simply a logic fitting problem where a logic function is extracted from a cone consisting of X k-input LUTs and then is checked if it can fit into a programmable structure containing less than X k-input LUTs. This can be solved using our QSAT technique.

To illustrate this process, consider Fig. 4. The original cone 4a consists of three 2-LUTs which implements a three input function. Since only three inputs enter the cone, it may be possible to resynthesize 4a into 4b to save one LUT.



(a) Original Cone (b) Resynthesized Cone

Fig. 4. Resynthesis of three-input cone of logic

To determine if resynthesis from 4a to 4b is possible, 4b is converted into a QBF and the function extracted from 4a is tested using QSAT to see if it fits into 4b. If the expression is satisfiable, resynthesis can occur.

4 Results

4.1 Technology Mapping to Apex20k PLB

The results of **SATMAP** for PLB technology mapping are shown here. The PLB of choice was the 5-input Altera Apex20k PLB which consists of a 4-LUT

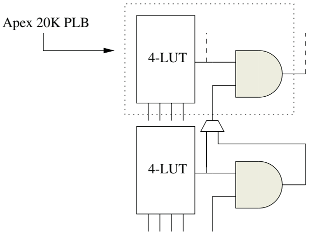


Fig. 5. Two Apex20k PLBs with routing constraints shown

feeding into a 2-input AND gate. A simplified view, shown in Fig. 5, is used in our experiments. Although a simplified PLB was used, routing constraints were maintained. These constraints required the PLB AND-input to come from an adjacent PLB as illustrated in Fig. 5.

Tab. 4.1 shows the total depth and area costs for the largest 20 MCNC benchmark circuits. The results clearly show that **SATMAP** [2] is an effective tool for technology mapping to PLBs as it outperforms any 4-LUT technology mapper. For depth the unit delay model is used and for area cost each 4-LUT is given a cost of 1. Area cost uses the assumption that the area of an AND gate is insignificant compared to the area of a 4-LUT. The upper table show results when depth was the primary optimizing goal and the lower table show results when area was the primary optimizing goal. The *Ratio* is the *Total* ratio when compared against **SATMAP** for each respective goal.

Table 1. Comparing **SATMAP** to IMap, FlowMap-r3, and ZMap with $k = 4$

	Depth Orientated $k = 4$							
	SATMAP		*IMap*		*FlowMap-r0*		*ZMap*	
	Area	Depth	Area	Depth	Area	Depth	Area	Depth
Total	35073	203	37856	206	42219	204	39043	203
Ratio	1.000	1.000	1.079	1.015	1.204	1.005	1.113	1.000

	Area Orientated $k = 4$							
	SATMAP		*IMap*		*FlowMap-r3*		*ZMap*	
	Area	Depth	Area	Depth	Area	Depth	Area	Depth
Total	33364	298	34859	307	35950	243	35883	295
Ratio	1.000	1.000	1.045	1.030	1.078	0.815	1.076	0.990

4.2 Resynthesis Results

For the resynthesis application, we performed resynthesis on circuits produced by the ZMap techmapper — one of the best publically available FPGA area-driven

[2] The applications presented in this paper were incorporated with the Berkeley MVSIS project [3].

techmappers developed by J. Cong et al. at UCLA [4]. Given a set of circuits, we used ZMap to technology map these circuits to **4**-LUTs. After some post processing done by RASP [4] to further improve area, we ran our resynthesizer on these LUT networks.

The number of resynthesis structures is countless; however, considering that the runtime of QSAT is exponential to the number of resynthesis structure inputs, for practical purposes, our work dealt with cones of fanin size 10 or less. Since we decided to remove quantifiers in our QBF, resynthesis checking was done using the Chaff SAT solver developed by M. W. Moskewicz et al. [9].

Benchmark Circuits. In our first set of resynthesis experiments, we focused on a set of circuits taken from the MCNC and ITC'99 benchmark suites ([11],[5]). These circuits were optimized using SIS [10] and RASP, technology mapped with ZMap, and resynthesized with our work. The optimization in SIS is particularly important since the structure of the gate-level netlist can have a significant impact on the mapped area. The left hand table in Tab. 4.2 shows the results. The *ZMap* column indicates the number of 4-LUTs the circuit was technology mapped to. The *Resynth* column indicates the number of 4-LUTs after our resynthesis.

Building Block Circuits. In our second set of resynthesis experiments, we focused on common digital circuit logic blocks. We started from Verilog code, synthesized it using VIS [2], then optimized and techmapped the circuits as in the benchmark circuit section.

The right hand table in Tab. 4.2 shows our results, where we achieve a reduction as large as 67% and an average reduction of 26%. Since these logic blocks are common in digital circuits, heuristics can be used to identify them and technology map them to our optimized circuits.

Table 2. Resynthesis results for benchmark circuits and common logic blocks

Circuit	*Resynth*	*ZMap*	*Ratio*
clma	4792	5014	0.95
b15_1	4112	4291	0.95
b15_1_opt	3772	3879	0.97
s38584.1	3454	3771	0.91
s38417	3444	3586	0.96
b14	2902	3072	0.94
frisc	2571	2624	0.98
pdc	1875	1928	0.97
misex3	1156	1184	0.98
seq	1162	1182	0.98
alu4	1103	1129	0.98
ex5p	968	993	0.97
i10	764	789	0.97
Total	32075	33442	0.96

Building Block	*Resynth*	*ZMap*	*Ratio*
4:1 MUX	2	3	0.67
16:1 MUX	21	29	0.72
32-Bit Priority Encoder	59	74	0.80
4-Bit Barrel Shifter	8	12	0.67
16-Bit Barrel Shifter	32	48	0.67
6-Bit Set Reset Checker	2	3	0.67
2-Bit Sum Compare Const	2	6	0.33
2-Bit Sum Compare	2	3	0.67
6-Bit Priority Checker	3	6	0.50
8-Bit Bus Multiplexor	16	24	0.67
Total	188	253	0.74

5 Conclusion and Future Work

In this work, we have shown a practical application of Quantified Boolean Satisfiability to the problem of cost reduction of FPGA-based circuit realizations. First we presented a generic technology mapping that is capable of mapping a circuit description to any arbitrary logic block. This allows for the study of architectural features that can be used to reduce the number of PLBs required to implement a circuit. In particular, we have shown that the use of an extra dedicated AND-gate can lead to significant area reductions in many cases; Second, we presented a method to reduce the number of k-input LUTs required to implement subcircuits for LUT-based FPGA architectures. We have shown area reductions of up to 67% in some cases using these techniques. This technique can be used to find other optimal configurations for small digital circuits which can be used in a step after technology mapping to replace logic blocks with their optimal configuration.

References

1. Altera. Component selector guide ver 14.0, 2004.
2. R. K. Brayton and G. D. H. et al. VIS: a system for verification and synthesis. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, pages 428–432, 1996.
3. D. Chai, J. Jiang, Y. Jiang, Y. Li, A. Mishchenko, and R. Brayton. MVSIS 2.0 Programmer's Manual, UC Berkeley. Technical report, Electrical Engineering and Computer Sciences, University of California, Berkeley, 2003.
4. J. Cong, J. Peck, and Y. Ding. RASP: A general logic synthesis system for SRAM-based FPGAs. In *FPGA*, pages 137–143, 1996.
5. F. Corno, M. Reorda, and G. Squillero. RT-level ITC 99 benchmarks and first ATPG results, 2000.
6. T. Larrabee. Test Pattern Generation Using Boolean Satisfiablity. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, 1992.
7. A. Ling. Field-Programmable Gate Array Logic Synthesis using Boolean Satisfiability. Master's thesis, University of Toronto, Toronto, ON, Canada, 2005.
8. V. Manohararajah. *Area Optimizations in FPGA Architecture and CAD.* PhD thesis, University of Toronto, Toronto, ON, Canada, 2005.
9. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
10. E. M. Sentovich, K. J. Singh, C. M. L. Lavagno, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, Electrical Engineering and Computer Sciences, University of California, Berkeley, 1992.
11. S. Yang. Logic synthesis and optimization benchmarks user guide version, 1991.

On Applying Cutting Planes in DLL-Based Algorithms for Pseudo-Boolean Optimization

Vasco Manquinho and João Marques-Silva

Technical University of Lisbon,
IST/INESC-ID, Lisbon, Portugal
{vmm, jpms}@sat.inesc-id.pt

Abstract. The utilization of cutting planes is a key technique in Integer Linear Programming (ILP). However, cutting planes have seldom been applied in Pseudo-Boolean Optimization (PBO) algorithms derived from the Davis-Logemann-Loveland (DLL) procedure for Propositional Satisfiability (SAT). This paper proposes the utilization of cutting planes in a DLL-style PBO algorithm, which incorporates the most effective techniques for PBO. We propose the utilization of cutting planes both during preprocessing and during the search process. Moreover, we also establish conditions that enable clause learning and non-chronological backtracking in the presence of conflicts involving constraints generated by cutting plane techniques. The experimental results, obtained on a large number of classes of instances, indicate that the integration of cutting planes with backtrack search is an extremely effective technique for PBO.

1 Introduction

In this paper we address algorithms for Pseudo-Boolean Optimization (PBO) and focus on exploiting effectively the information provided by the cost function. Our objective is to use this information for pruning the search space. Hence, we propose to augment SAT-based PBO algorithms with bounding capability, associated with information obtained from the Pseudo-Boolean (PB) constraints and from the cost function[1]. Moreover, we propose to extend a SAT-based PBO algorithm with lower bounding, that uses linear programming relaxation for compute lower bounds [2], and integrate the identification of cutting planes in this algorithm. We also establish conditions for learning new constraints from conflicts associated with cutting planes. Furthermore, we show that these new constraints can be used for performing non-chronological backtracking. Experimental results, obtained on representative problem instances, illustrate the effectiveness of integrating cutting planes in SAT-based algorithms for PBO.

[1] An extended version of this paper is available in [1].

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 451–458, 2005.

2 Preliminaries

An instance P of the Pseudo-Boolean Optimization problem can be defined as follows:

$$\begin{array}{ll} \text{minimize} & \sum\limits_{j \in N} c_j \cdot x_j \\ \text{subject to} & \sum\limits_{j \in N} a_{ij} l_j \geq b_i, \\ & x_j \in \{0,1\}, a_{ij}, b_i \in \mathbf{N}_0^+, i \in M \\ & N = \{1, \ldots, n\}, M = \{1, \ldots, m\} \end{array} \tag{1}$$

where c_j is a non-negative integer cost associated with variable $x_j, j \in N$ and a_{ij} denote the coefficients of the literals l_j in the set of m linear constraints.

Every pseudo-boolean formulation can be rewritten such that all coefficients a_{ij} and right-hand side b_i be non-negative. In a given constraint, if all a_{ij} coefficients have the same value k, then it is called a cardinality constraint, since it only requires that $\lceil b_i/k \rceil$ literals be true. A pseudo-boolean constraint where any literal set to true is enough to satisfy the constraint, can be interpreted as a propositional clause. This occurs when the value of all a_{ij} coefficients are greater than or equal to b_i. If every constraint can be interpreted as a propositional clause then P is an instance of the *binate covering problem* (BCP). Covering formulations have been the subject of thorough research work [3].

Notice that a linear pseudo-boolean optimization problem can also be viewed as a special case of linear integer programming problem. The linear integer programming formulation for the constraints can be obtained if we replace literals $\bar{x}_j$ by $1 - x_j$. Throughout the paper we refer extensively to backtrack search algorithms. In addition, the PB inference techniques of [4] are assumed.

3 Pseudo-Boolean Optimization Algorithms

Given that PBO is a restriction of generic ILP, all algorithms proposed in the past for ILP can also be used for PBO. Among these, complete approaches include branch-and-bound with linear programming relaxations [5], cutting planes [6], branch-and-cut [7], and branch-and-bound [3]. Besides algorithms for generic ILP, algorithms specific to PBO have also been proposed. These include SAT-based algorithms [4, 8, 9], branch-and-bound algorithms [3] and SAT-based algorithms with lower bounding [2]. A survey of these algorithms is available, for example, in [1]. In the reminder of this section we address the utilization of linear programming relaxations.

Linear programming relaxations (LPR) have long been used as a lower bound estimation procedure in branch-and-bound algorithms for solving integer programming problems [5, 7, 10]. It is also often the case that the LPR bound is tighter than the one obtained through other lower bounding procedures [2, 11]. The general formulation of the LPR for a pseudo-boolean problem instance is obtained from (1) as follows:

$$\begin{array}{ll}\text{minimize} & z_{lpr} = cx \\ \text{subject to} & Ax \geq b,\ 0 \leq x \leq 1\end{array} \tag{2}$$

where vector c defines the non-negative integer cost associated with every decision variable in vector x. Entries of matrix A define the constraint coefficients and vector b the right-hand side of every constraint. It is well-known that the solution of (2) is a lower bound on the solution of (1) [5].

4 Cutting Planes

Integer Linear Programming algorithms use linear programming relaxations (as formulated in (2)) for estimating lower bounds on the value of the cost function. However, linear programming relaxations have other applications, including the identification of cutting planes.

The work on cutting planes can be traced to Gomory [6]. Gomory introduced a cutting plane technique that derives new linear inequalities in order to exclude some non-integer solutions from (2). However, the new linear inequalities are valid for the original integer linear program and so can be safely added to the original problem. Moreover, solving (2) with the added inequalities may yield a tighter lower bound estimate.

Since Gomory's original work, a large number of cutting plane techniques have been proposed [5, 10, 12]. This section addresses Gomory fractional cuts and clique cuts, which are integrated in a SAT-based PBO solver.

In simplex-based solutions for solving the LPR from (2), the simplex method adds a set S of slack variables such that each constraint can be formulated as:

$$\sum_{j \in N} a_{ij} x_j - s_i = b_i \ \ s_i \geq 0 \tag{3}$$

This formulation is called the slack formulation and it is used to create the original simplex tableau [5].

If the solution x^* of the LPR is integral, then x^* provides the optimal solution to the original problem. Otherwise, choose a basic[2] variable x_j such that its value on the LPR solution is not integral. Since x_j is a basic variable, after the pivot operations performed by the simplex algorithm on (3), there is a row in the simplex tableau of the form,

$$x_j + \sum_{i \in P} \alpha_i x_i + \sum_{i \in Q} \beta_i s_i = x_j^* \tag{4}$$

where P and Q are the sets of indexes of non-basic variables (problem variables and slack variables, respectively). In [6], Gomory proves that the inequality,

$$\sum_{i \in P} f(\alpha_i) x_i + \sum_{i \in Q} f(\beta_i) s_i \geq f(x_j^*) \tag{5}$$

[2] See for example [5] for a definition of basic and non-basic variables.

where $f(y) = y - \lfloor y \rfloor$, is violated by the solution of the LPR, but satisfied by all non-negative integer solutions to (4). Hence, it is also satisfied by all solutions to the original problem as formulated in (1) and can be added to the LPR. Solving the LPR with the new restriction (known as a Gomory *fractional cut*) will yield a tighter lower bound estimate on the value of the PBO instance. Observe that several methods for strengthening the original Gomory fractional cuts have been proposed [13, 14, 15], but are beyond the scope of this work.

Notice that we can use (3) to replaces each slack variable in (5) so that the constraint only contain variables from the original PBO problem. Afterwards, if we apply the rounding operation on the non integer coefficients we obtain a new pseudo-boolean constraint valid for the original PBO instance as defined in (1), since the rounding operation will only weaken the constraint.

Like the Gomory fractional cuts, clique cuts [5, 10] also provide a method that adds new inequalities in order to cut non-integral solutions from the LPR, hence improving the tightness of lower bound estimates.

In general, we can build a conflict graph in order to represent all incompatible assignments for a pseudo-boolean formula. In the conflict graph, each node represents an assignment to a problem variable and each edge between two nodes represents an assignment incompatibility. For each clique C in the conflict graph we can add a new constraint of the form $\sum_{i \in C} l_i \leq 1$, where l_i is the literal at node i of clique C. One should note that we are interested in finding all maximum cliques in the conflict graph, but it is well-known that that the problem of finding a maximum clique in an undirected graph is NP-Hard [5]. As a result, a heuristic greedy procedure is often used.

5 Cutting Planes in SAT-Based PBO

In a modern SAT-based algorithm, a conflict analysis procedure is carried out whenever a conflict arises [16]. Therefore, if the generated cutting plane is involved in the conflict analysis process, it must be able to determine its logical dependencies in order to backtrack to a valid node of the search tree. This section proposes conditions for associating dependencies with computed cutting planes, thus enabling clause learning and non-chronological backtracking from constraints inferred with cutting plane techniques.

The most straightforward solution, for safely determining a set of dependencies for the Gomory fractional cuts generated during the search process, is to declare that these cuts depend on all decision assignments made from the root node to the current node. This solution associates with each cutting plane all decisions in the search tree, thus forcing chronological backtracking and ensuring completeness. In this case, we can determine a set of literals ω_{cut} that defines the set of dependencies for the generated cut. When one of literals in ω_{cut} is set to 1, the cut will no longer be active (i.e. the associated constraint will be satisfied). Therefore, the generated cut would depend on all decision assignments and, for all decision variables x_j assigned from the root node to the current node, we would have $\bar{x}_j \in \omega_{cut}$ if $x_j = 1$ or $x_j \in \omega_{cut}$ if $x_j = 0$.

In order for the generated cut to be safely added to the set of pseudo-boolean constraints, we must add all literals $l_j \in \omega_{cut}$ to the cut. The coefficient of each added literal l_j must be large enough to satisfy the constraint whenever $l_j = 1$. Although this approach guarantees the completeness of the algorithm, if a conflict occurs involving the generated cuts at a node N of the search tree, the search cannot backtrack to a node higher than N.

Another technique would be to associate dependencies with cuts following the ideas proposed in [2] for LPR. Since each cut is derived from the outcome of solving the LPR formulation, then we can associate with each cut the same dependencies we associate with lower bound conflicts. However, one should note that the tableau constraint (4), from which the Gomory fractional cut is inferred, depends on the pivot operations performed while solving the LPR. As a result, the tableau constraint (4) contains the slack variables assigned value 0 from the constraints from which it depends.

Let S be the set of constraints with slack variables assigned value 0 in the tableau constraint (4). If the literals assigned value 0 in these constraints were to have a different value, the tableau constraint might not be inferred in the LPR. Therefore, we can consider the assignments to those literals as the responsible for inferring the cut and we can define ω_{cut} as:

$$\omega_{cut} = \{l : l = 0 \wedge l \in \omega_i \wedge \omega_i \in S\} \tag{6}$$

Notice that the generated cut might not depend on all decision assignments. Hence, if a conflict occurs involving generated cuts at node N with its dependences determined as in (6), it is possible to backtrack to a node higher than N in the search tree, i.e. a non-chronological backtrack step. Moreover, the generated cuts can also be used in different parts of the search tree, in addition to the subtree with root at the node N.

6 Results

In order to empirically evaluate the techniques described in the paper, we ran our solver (*bsolo*) on representative sets of PBO instances [17, 18, 19]. Besides bsolo, we also ran PBS [8], Galena [4] and the commercial MILP solver CPLEX (version 7.5). The CPU times presented are from a AMD Athlon processor at 1.9 GHz and the time limit for each instance was set to one hour. If the time limit was reached, we provide an indication of which was the best upper bound value found when the search was stopped. Additional results and details on the experimental procedure can be found at [1].

The experimental results are shown in Table 1. For bsolo we present results for four different configurations: without using cutting planes, using only fractional Gomory cuts during the search, using clique cuts and Gomory fractional cuts only during preprocessing, and using all cuts both during preprocessing and during the search. For all bsolo configurations, the lower bound estimates were obtained with linear programming relaxations.

Table 1. Experimental Results

						bsolo			
	Benchmark	sol.	pbs	galena	cplex	no cuts	Gomory	pre proc.	all cuts
[17]	9symml	4517	ub6453	ub6986	1.63	328.97	41.39	716.56	225.62
	C432	4822	ub6577	ub8070	3.34	ub4822	343.33	1253.60	208.52
	cmb	1053	ub1490	ub1476	0.03	0.25	0.23	0.12	0.10
	my_adder	4561	ub6271	ub5548	2.29	14.94	6.54	10.15	5.07
[18]	9sym.b	5	1718.98	0.26	0.14	0.89	0.68	0.73	0.72
	alu4.b	–	ub121	ub53	ub50	ub50	ub50	ub51	ub50
	apex4.a	776	ub2282	ub845	3.92	ub776	810.22	ub776	2404.60
	clip.b	15	ub30	1.68	0.36	9.65	1.15	4.14	5.54
	e64.b	–	ub99	ub53	ub49	ub50	ub49	ub50	ub49
	jac3	15	ub48	0.71	0.09	3.11	2.93	9.67	6.21
	rot.b	115	ub745	ub142	71.56	ub117	ub117	ub118	ub118
	sao2.b	25	ub39	132.91	0.50	11.3	5.46	9.34	18.99
[19]	aim100-1_6-y1-2	100	0.01	0.01	373.42	0.04	0.04	0.19	0.19
	aim100-3_4-y1-4	100	0.01	–	108.79	2.73	2.76	0.30	0.31
	aim200-1_6-y1-3	200	0.01	–	–	0.11	0.11	0.49	0.51
	aim200-6_0-y1-2	200	0.01	0.03	ub200	140.18	119.81	0.81	0.84
	ii8a2	139	ub150	ub144	63.77	ub141	ub141	1931.50	ub139
	ii8b1	191	ub272	ub213	7.11	2285.50	ub232	860.37	ub193
	ii8c1	302	ub405	ub399	2719.59	ub416	ub413	ub320	ub321
	ii8d1	–	ub515	ub432	ub351	ub470	ub470	ub359	ub401
	jnh1	92	0.01	0.20	39.56	42.81	324.99	35.84	158.63
	jnh7	89	0.02	0.04	7.42	20.55	9.80	1.91	3.04
	ssa7552-159	1327	ub1327	ub1327	ub1327	ub1327	ub1327	ub1327	1212.50
	ssa7552-160	1359	ub1359	ub1359	ub1359	ub1359	ub1359	ub1359	1369.60

Among the SAT-based PBO algorithms, bsolo is by far the most effective algorithm, for the instances in this paper and for the instances in [2]. In fact, bsolo without the utilization of cutting planes is already significantly more effective than the other SAT-based algorithms. The utilization of cutting planes further improves the results of bsolo, making it more robust than the commercial ILP solver cplex. Observe that the worst results for cplex occur for instances of the minimum-size prime implicant problem. These instances are derived from CNF formulas, where SAT-based techniques are particularly relevant.

7 Conclusions

The paper describes the integration of cutting plane techniques in SAT-based algorithms for Pseudo-Boolean Optimization and outlines conditions for performing constraint learning and non-chronological backtracking based on previously inferred cutting planes. These conditions provide novel mechanisms for extending the most effective SAT techniques to PBO. Experimental results clearly indicate that the utilization of cutting plane techniques can be extremely effective in PBO. Moreover, the results are also clear in demonstrating that lower bounding techniques are essential for hard instances of PBO. The experimental results for the most well-known PBO solvers that do not integrate lower bounding techniques, clearly demonstrate that lower bounding is essential for representative instances of pseudo-boolean optimization.

References

1. Manquinho, V., Marques-Silva, J.: On applying cutting planes in dll-based algorithms for pseudo-boolean optimization. Technical Report RT/003/05-CDIL, INESC-ID (2005)
2. Manquinho, V., Marques-Silva, J.P.: Effective lower bounding techniques for pseudo-boolean optimization. In: Design, Automation and Test in Europe Conference. (2005)
3. Coudert, O.: On Solving Covering Problems. In: Design Automation Conference. (1996) 197–202
4. Chai, D., Kuehlmann, A.: A Fast Pseudo-Boolean Constraint Solver. In: Design Automation Conference. (2003) 830–835
5. Nemhauser, G.L., Wolsey, L.A.: Integer and Combinatorial Optimization. John Wiley & Sons (1988)
6. Gomory, R.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society **64** (1958) 275–278
7. Mitchell, J.: Branch-and-cut algorithms for combinatorial optimization problems. In: Handbook of Applied Optimization. Oxford University Press (2002) 65–77
8. Aloul, F., Ramani, A., Markov, I., Sakallah, K.: Generic ILP versus specialized 0-1 ILP: An update. In: International Conference on Computer Aided Design. (2002) 450–457
9. 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)
10. Bixby, R.E.: Progress in linear programming. ORSA Journal on computing **6** (1994) 15–22
11. Liao, S., Devadas, S.: Solving Covering Problems Using LPR-Based Lower Bounds. In: Design Automation Conference. (1997) 117–120
12. Chvátal, V.: Edmonds polytopes and a hierarchy of combinatorial problems. Discrete Mathematics **4** (1973) 305–337
13. Balas, E., Ceria, S., Cornuéjols, G., Natraj, N.: Gomory cuts revisited. Operations Research Letters **19** (1996) 1–9
14. Ceria, S., Cornuéjols, G., Dawande, M.: Combining and strengthening Gomory cuts. In Springer-Verlag, ed.: Lecture Notes in Computer Science. Volume 920. E. Balas and J. Clausen (eds.) (1995)
15. Gomory, R.: An algorithm for integer solutions to linear programs. In Graves, R., (eds.), P.W., eds.: Recent Advances in Mathematical Programming. McGraw-Hill (1963) 269–302
16. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A new search algorithm for satisfiability. In: International Conference on Computer-Aided Design. (1996) 220–227

17. Zhu, Z.: Synthesis for mixed ptl/cmos circuit. (http://www-unix.ecs.umass.edu/-~zzhu/)
18. Yang, S.: Logic Synthesis and Optimization Benchmarks User Guide. Microelectronics Center of North Carolina (1991)
19. Pizzuti, C.: Computing Prime Implicants by Integer Programming. In: IEEE International Conference on Tools with Artificial Intelligence. (1996) 332–336

A New Set of Algebraic Benchmark Problems for SAT Solvers

Andreas Meier[1] and Volker Sorge[2]

[1] DFKI GmbH, Saarbrücken, Germany
ameier@dfki.de
[2] School of Computer Science, University of Birmingham, UK
V.Sorge@cs.bham.ac.uk

Abstract. We propose a new benchmark set consisting of problems generated during the construction of classification theorems for quasigroups. It extends and generalises the domain of quasigroup existence problems, to which SAT solvers have been applied successfully in the past, to a rich class of benchmarks of varying difficulty.

1 Introduction

Classification of mathematical structures is a challenging task in mathematical research. A first step in producing algebraic classification theorems is to determine for which sizes certain algebras exist. Automated support for solving such existence problems has been remarkably successful in the past. In particular, model generators [4, 8] and satisfiability (SAT) solvers [15, 14] have been successfully applied to show the existence or non-existence of quasigroups with particular associated properties. The study and discovery of quasigroups with certain properties is interesting also outside of pure mathematics, because the underlying structure of quasigroups is similar to that found in many real-world applications [6, 7]. While quasigroup existence problems present a challenging task for SAT solvers their use as a benchmark set is restricted as, firstly, the number of problems that can be tackled realistically by existing systems is relatively small and, secondly, the structure of the problems is relatively similar.

In [3] we have extended quasigroup existence problems to the more general classification problem: How can different isomorphism classes of quasigroups of a given cardinality be described by their algebraic properties? The reasoning problems that need to be solved when answering this question are generally of propositional nature and can effectively be transformed into SAT problems [9].

In this paper, we present a uniform view of the classification problems as generalised quasigroup existence problems and suggest them as a new set of benchmarks from an algebraic domain for the testing and development of SAT solvers. The classification process can be seen as generator for this new benchmark set containing problems of varying structure and complexity (Sec. 2). We describe some of the characteristics of the problems with respect to their encoding in propositional logic (Sec. 3) and present a non-trivial example to give an impression of the mathematics behind the domain (Sec. 4).

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 459–466, 2005.

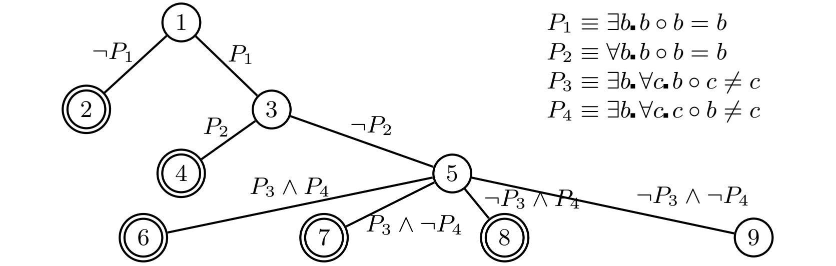
2 Quasigroup Classification Problems

In [3] we have presented a bootstrapping algorithm to automatically generate classification theorems in finite algebra. To guarantee correctness of the classification a number of theorems has to be shown. Each theorem is essentially an existence problem and can be tackled by SAT solvers. We can therefore view the overall bootstrapping algorithm as a generator of our benchmark set.

The *general existence problem in finite algebra* is the question whether for some cardinality n an algebra exists that satisfies a given set of axioms $\mathcal{A}$. A considerable amount of research in automated reasoning has been devoted to solving some existence problems for the particular domain of quasigroups. A quasigroup is a non-empty set Q together with a binary operation $\circ$ that satisfies the property $\forall a, b \in Q\text{.}\ (\exists x \in Q\text{.}\, x \circ a = b) \wedge (\exists y \in Q\text{.}\, a \circ y = b)$. This property is often called Latin Square property and has the effect that every element of Q appears exactly once in every row and every column of the multiplication table of $\circ$. One existence problem for quasigroups is, for instance, to ask whether a quasigroup of a given cardinality n exists for which the operation $\circ$ also satisfies the QG4-property $\forall x, y \in Q\text{.}\ (x \circ (x \circ y)) \circ y = x$. SAT solvers could show, for example, that a QG4-quasigroup of cardinality $n = 14$ exists [15].

A *classification problem in finite algebra* for a given cardinality n and a set of axioms $\mathcal{A}$ is the question: How many different algebras exist that satisfy $\mathcal{A}$ and how can they be described? By "different" we mean here that the algebras are not isomorphic to each other; thus the classification problem is actually concerned with the detection and the axiomatic description of all isomorphism classes of algebras satisfying the axioms $\mathcal{A}$. In [3] we tackle this problem employing a bootstrapping algorithm. It automatically constructs a decision tree by successively identifying non-isomorphic structures and their discriminating properties until all possible isomorphism classes are generated and for each isomorphism class a *representant* (i.e., an algebra that is element of the isomorphism class) is found. A property P acts as a *discriminant* for any two algebras A and B iff $P(A)$ and $\neg P(B)$ means that A and B are not isomorphic. An isomorphism class can be uniquely described by its associated discriminants that form a *classifying property*, that is a property that holds for every algebra in the isomorphism class, but does not hold for any algebra outside the isomorphism class.

For example, Fig. 1 contains the decision tree for the classification problem of order 3 quasigroups together with the representants for each of the 5 isomorphism classes, where representant A_i belongs to the node i in the tree. The edges of the decision tree are labelled with discriminants and each node in the tree is associated with a *describing property*, which is the conjunction of all discriminants along the path from the root to the node. The nodes associated with isomorphism classes are the leaf nodes marked with a double circle. The classifying property of an isomorphism class is the describing property of the associated node. For instance, the isomorphism class represented by node 6 has the classifying property of $P_1 \wedge \neg P_2 \wedge P_3 \wedge P_4$. If during the construction of the decision tree a node is reached for which no representant exists that satisfies the


1
$\neg P_1$
P_1
2
3
P_2
$\neg P_2$
4
5
$P_3 \wedge P_4$
$\neg P_3 \wedge P_4$
$\neg P_3 \wedge \neg P_4$
$P_3 \wedge \neg P_4$
6
7
8
9
$P_1 \equiv \exists b. b \circ b = b$
$P_2 \equiv \forall b. b \circ b = b$
$P_3 \equiv \exists b. \forall c. b \circ c \neq c$
$P_4 \equiv \exists b. \forall c. c \circ b \neq c$

A_2	a	b	c
a	b	a	c
b	a	c	b
c	c	b	a

A_4	a	b	c
a	a	c	b
b	c	b	a
c	b	a	c

A_6	a	b	c
a	a	b	c
b	b	c	a
c	c	a	b

A_7	a	b	c
a	a	b	c
b	c	a	b
c	b	c	a

A_8	a	b	c
a	a	c	b
b	b	a	c
c	c	b	a

Fig. 1. Decision tree and isomorphism class representants for order 3 quasigroups

describing property of the node, then this node is a dead-end in the tree. Node 9 in the tree in Fig. 1 is an example of such a node.

To *verify the correctness of the classification* the properties of the decision tree have to be proved. Thereby, the most difficult problem is to show that actually a leaf node is reached, i.e., that a node either represents an isomorphism class or that it is a dead-end. In the latter case we show that no algebra of cardinality n satisfies the describing property of the node (we call this the Dead-End Theorem), which is obviously an existence problem. In the former case we have to show that all algebras of cardinality n that satisfy the describing property of the node are isomorphic (Isoclass Theorem). This is also an existence problem, since it states that there does not exist an algebra A that satisfies the describing property and is not in the isomorphism class.

Although the isoclass theorems are essentially second order, they can be expressed as propositional logic problems by enumerating all possible isomorphism functions for structures of cardinality n. The naïve approach, considering all $n!$ possible isomorphisms, can be improved upon by first proving a lemma that states that all algebras with cardinality n satisfying a property P share a generating system[1]. Since generating systems are invariant under isomorphism (i.e., if algebra A has a generating system and algebra B is isomorphic to A, then B has also such a generating system) it is a necessary prerequisite that this lemma holds for the isomorphism-class theorem to hold. We call this lemma the *generating-system lemma*. If it can be shown, then the number of isomorphisms for the isoclass theorem can often be drastically reduced to only the possible mappings of the generators. The generating-system lemma is again an existence problem stating that there does not exist an algebra A that satisfies classifying property P, but does not exhibit a certain generating system.

[1] A structure A with binary operation $\circ$ is said to be *generated* by a set of elements $\{a_1, \ldots, a_m\} \subseteq A$ if every element of A can be expressed as a combination – usually called a factorisation or word – of the a_i under the operation $\circ$. We call a set of generators together with the corresponding factorisations a *generating system*.

3 The Benchmark Set and Its Characteristics

The *problem set* consists of a collection of generating-system lemmas, isoclass theorems, and dead-end theorems for quasigroups of cardinality 6, 7, and 8. The problems have been generated from decision trees for the classification of

- non-idempotent quasigroups of cardinality 6 with a simplified version of the QG3 property: $\exists a_{\bullet}\forall b_{\bullet}(b \circ a) \circ (a \circ b) = b$,
- quasigroups 7 with the QG9 property: $\forall a, b, x, y_{\bullet} a \circ b = x \land x \circ b = y \to y \circ b = a$,
- quasigroups 8 with a simplified version of QG9: $\forall a, b, x_{\bullet} a \circ b = x \to x \circ b = a$.

The current set contains roughly 1000 problems and has been submitted in the DIMACS format as a benchmark set to the SAT05 system competition. The set comprises both satisfiable and unsatisfiable problems. This has essentially two reasons: Firstly, during the construction of the decision tree, generating-system and isoclass theorems are only shown for leaf nodes associated with isomorphism classes. If the tree has been constructed correctly these theorems hold and the corresponding SAT problems are unsatisfiable. However, for the benchmark set we also conjecture generating-system lemmas and isoclass theorems for branching nodes, with respect to the describing properties of that node. For a branching node, however, either one or both of the conjectures can not hold, which leads to satisfiable problems. Secondly, the decision trees for the quasigroups 7 and 8 problems are so far only partially constructed and therefore the status of some of the resulting SAT problems is not yet known.

Our *encoding* follows Zhang's approach [13], where a boolean variable corresponds to an equation of the form $x \circ y = z$ or $x = y$, where x, y, z are instantiated with the n elements of the algebra of cardinality n. Hence, to encode all possible equations $n^3 + n^2$ boolean variables are necessary, such that the number of boolean variables increases wrt. the cardinality n (see [9] for details).

Since the discriminant properties can be arbitrarily complex and nested it turns out that a naïve clause normalisation approach suffers in our domain from a combinatorial explosion of the number and the length of the resulting clauses. Hence, we adopted clause normalisation techniques from [11] that aim to create small clause normal forms. The basic technique is the introduction of additional boolean variables to suitably break formulas. This avoids combinatorial explosion and restricts the size of the resulting clauses but extends the problem formalisation by additional boolean variables. Our clause normalisation breaks formulas when the resulting clauses become larger than $3{\cdot}n$. The introduction of additional boolean variables to break the formulas, however, can result in final clauses that are slightly larger than $3{\cdot}n$. The optimal clause normalisation for our problems is still subject to testing.

The *difficulty* of propositional satisfiability problems can generally be characterised both by the number of boolean variables as well as by the number and sizes of the formulas or clauses involved. In our domain, both the number of variables and clauses increase for larger cardinalities n and for nodes deeper in the decision tree. To illustrate this consider the figures characterising

Table 1. Characteristic numbers of quasigroup generating-system lemma problems

Cardinality	depth of node	#variables	#clauses	#max clause size
5	1	150	2055	5
5	23	352	4531	9
6	1	257	5925	6
6	7	405	6975	8
7	1	2401	20594	7
7	8	4044	24907	12
8	1	576	53212	7
8	8	12671	110508	28

generating-system lemmas of quasigroup classification problems with cardinality 5–8 in Table 1.

We have applied the SAT solvers zChaff [10], DPLLT [5], CVClite [2] and the first-order theorem prover Spass [12] to different formalisations of our benchmark problems (see [9] for details). These experiments reveal that on average the difficulty of the problems does indeed increase with increasing numbers of variables and clauses. However, they also show that there is a high variance in the time needed by the systems to solve problems of roughly the same size. While some problems could be solved very quickly, other problems with a similar number of variables and clauses would take very long to solve or could not be solved at all. In fact, the complexity of the clauses for a problem in our domain depends mainly on the form of the describing property P, which is a conjunction of discriminant properties (see Sec. 2). In general, P is more complex the deeper the corresponding node is in the tree. However, the complexity of the discriminants of which P is composed can differ (see the example in the next section) and the difficulty they pose to SAT solvers can vary. For instance, the discriminant $\forall b. b \circ b = b$ simplifies the problem at hand as opposed to a property like the quasigroup property itself, which makes the problem generally more difficult.

For similar reasons isoclass theorems are generally considerably less difficult than dead-end theorems and generating-system lemmas. The usage of the generating-system lemma as axiom in the isoclass theorem drastically reduces the complexity of this theorem, since the generating-system lemma results in unit clauses restricting the search. And our experiments indeed established the fact that the generating-system lemmas are the hardest problems of our domain.

4 An Example Problem

To give an impression of the mathematical quality of the problems in our benchmark set we present a non-trivial example from our domain. For the classification of quasigroups of order 7 with the QG9 property our algorithm computes an isomorphism class described by the representant A below together with a classifying property consisting of the conjunction of the following 11 properties:

A	e_1	e_2	e_3	e_4	e_5	e_6	e_7
e_1	e_4	e_1	e_5	e_7	e_6	e_2	e_3
e_2	e_1	e_6	e_3	e_4	e_5	e_7	e_2
e_3	e_7	e_4	e_6	e_2	e_1	e_3	e_5
e_4	e_2	e_7	e_1	e_3	e_4	e_5	e_6
e_5	e_3	e_2	e_4	e_5	e_7	e_6	e_1
e_6	e_6	e_5	e_2	e_1	e_3	e_4	e_7
e_7	e_5	e_3	e_7	e_6	e_2	e_1	e_4

1. $\forall b. b \circ b \neq b$
2. $\exists b. \forall c. b \circ c \neq c$
3. $\exists b, c. b \circ c = b \wedge c \circ b = b$
4. $\forall b. \exists c, d. c \circ d = b \wedge d \circ c \neq b$
5. $\exists b. \forall c. (c \circ b) \circ c \neq b$
6. $\exists b. \forall c. b \circ (b \circ c) \neq c$
7. $\forall b. \exists c. b \circ (b \circ c) \neq c$
8. $(\exists b. \forall c. (b \circ c) \circ (b \circ c) \neq c) \wedge$
   $(\forall b, c. b \circ b \neq c \vee c \circ c \neq b)$
9. $\exists b, c. b \circ b = c \wedge b \circ c = b$
10. $\exists b. \forall c. (b \circ c) \circ b \neq c$
11. $\exists b. \forall c. c \circ (c \circ b) \neq b \vee b \circ c = c \circ b$

For the representant A the algorithm computes a generating system, whose set of generators consists of e_7 only and the factorisations are:

$$e_1 = ((e_7 \circ e_7) \circ (e_7 \circ e_7)) \circ (((e_7 \circ e_7) \circ (e_7 \circ e_7)) \circ e_7) \qquad e_4 = e_7 \circ e_7$$
$$e_2 = (e_7 \circ (e_7 \circ e_7)) \circ ((e_7 \circ e_7) \circ (e_7 \circ e_7)) \qquad e_5 = ((e_7 \circ e_7) \circ (e_7 \circ e_7)) \circ e_7$$
$$e_3 = (e_7 \circ e_7) \circ (e_7 \circ e_7) \qquad e_6 = e_7 \circ (e_7 \circ e_7)$$

The generating-system lemma for this isomorphism class states that all algebras of order 7 that satisfy the given classifying property exhibit a generating system of the above form, i.e., they exhibit exactly this generating system or a generating system resulting from the permutation of the elements $e_1, e_2, e_3, e_4, e_5, e_6, e_7$. This results in 7! concrete possible generating systems, which have to be encoded in propositional logic. The final SAT problem then consists of 972 boolean variables and 13573 clauses with a maximal clause length of 13 literals.

If the generating-system lemma is shown successfully, the corresponding isoclass theorem can be stated as: All algebras of order 7 that satisfy the given classifying property and have a generating system of the above form are isomorphic to A. The possible isomorphisms that need to be considered are now restricted to the set of generators. Since we only have one generator, there are 7 possible isomorphisms, which can be further restricted by discarding those mappings that would violate the homomorphism property (see [9] for details). For the problem at hand it turns out that there is exactly one possible isomorphism on the generating system that needs to be instantiated into the isoclass theorem. The concrete SAT problem then consists of 1122 boolean variables and 8662 clauses with a maximum clause length of 14 literals.

Although the pure numbers of variables and clauses might suggest that the isoclass theorem and the generating-system lemma are approximately of the same complexity, the actual complexity is hidden in the generating-system lemma (see discussion in previous section). And indeed when applying zChaff to the problems it proves the isoclass theorem in 0.1 seconds but needs 19154.9 seconds to show the generating-system lemma. The other systems we employed in our experiments exhibited the same drastic difference in performance.

5 Conclusions

We have presented the domain of classification problems for quasigroups and suggest it as a new set of benchmarks for satisfiability solving. We believe that using these problems for the development and testing of SAT solvers can be of mutual benefit: On the one hand, the SAT community will gain a large testbed of problems of varying difficulty and structure. On the other hand stronger and better SAT solvers can aid in the derivation of new mathematical classification results. While we have concentrated on quasigroups for the submitted benchmark set, our classification algorithm has already been applied to other algebraic structures, such as monoids and semi-groups, which might also be exploitable in the future.

The experience we have gained so far suggests that the successful application of SAT solvers in our domain relies on an effective clause normalisation and a suitable formalisation of properties. Indeed, in our experiments different formalisations of properties (e.g., the quasigroup property) had also a considerable impact on the difficulty of the resulting SAT problems. However, it is not yet clear how, in general, properties of our domain can be best reformulated or encoded to enhance the performance of a SAT solver. Further experiments with large numbers of properties should give us more insights into these questions. These might also prove beneficial for other, possibly non-mathematical domains.

References

1. R. Alur, D. Peled, editors. *Proc. of CAV-2004*, LNCS 3114. Springer, 2004.
2. C. Barrett, S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Alur and Peled [1], p.515–518.
3. S. Colton, A. Meier, V. Sorge, R. McCasland. Automatic generation of classification theorems for finite algebras. In *Proc. of IJCAR–2*, LNAI 3097, p.400–414. Springer, 2004.
4. M. Fujita, J. Slaney, F. Bennett. Automatic Generation of Some Results in Finite Algebra. *Proc. of IJCAI–13*, p.52–57. Morgan Kaufmann, 1993.
5. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, C. Tinelli. Dpll(t): Fast decision procedures. In Alur and Peled [1], p.175–188.
6. S.R. Kumar, A. Russel, R. Sundaram. Approximating latin square extensions. *Algorithmica*, 24:128–138, 1999.
7. C. Laywine, G. Mullen. *Discrete Mathematics using Latin Squares.* Wiley, 1998.
8. W.W. McCune. A davis-putnam program and its application to finite first-order model search: quasigroup existence problems. Report, Argonne Nat. Labs, 1994.
9. A. Meier, V. Sorge. Applying sat solving in classification in finite algebra. Submitted to the Journal of Automated Reasoning.
10. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of the Design Automation Conference*, p.530–535, 2001.
11. A. Nonnengart, C. Weidenbach. Computing small clause normal forms. In *Handbook of Automated Reasoning.* Elsevier, 2001.
12. C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, D. Topic. SPASS version 2.0. In *Proc. of CADE–18*, LNAI 2392, pages 275–279. Springer, 2002.

13. H. Zhang. Specifying latin squares in propositional logic. In *Automated Reasoning and Its Applications, Essays in honor of Larry Wos.* MIT Press, 1997.
14. H. Zhang, M. Bonacina, J. Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *J. of Symb. Computation*,21:543–560, 1996.
15. H. Zhang, J. Hsiang. Solving Open Quasigroup Problems by Propositional Reasoning. In *Proc. of International Computer Symposium*, 1994.

A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas

Maher Mneimneh[1], Inês Lynce[2], Zaher Andraus[1],
João Marques-Silva[2], and Karem Sakallah[1]

[1] University of Michigan,
{maherm, zandrawi, karem}@umich.edu
[2] Technical University of Lisbon, Portugal
{ines,jpms}@sat.inesc-id.pt

Abstract. We tackle the problem of finding a smallest-cardinality MUS (SMUS) of a given formula. The SMUS provides a succinct explanation of infeasibility and is valuable for applications that rely on such explanations. We present a branch-and-bound algorithm that utilizes iterative MAXSAT solutions to generate lower and upper bounds on the size of the SMUS, and branch on specific subformulas to find it. We report experimental results on formulas from DIMACS and DaimlerChrysler product configuration suites.

1 Introduction

Explaining the causes of infeasibility of Boolean formulas has practical applications in numerous fields: electronic design, formal verification, and artificial intelligence. In design applications, for example, a large Boolean function is formed such that a feasible design is obtained when the function is satisfiable, and design infeasibility is indicated when the function is unsatisfiable. An example of this is the routing of signal wires in an FPGA. We are usually interested in a "minimal" explanation of infeasibility that excludes irrelevant information. For Boolean formulas in conjunctive normal form (CNF), the notion of minimality is defined as follows. Consider an unsatisfiable CNF formula φ. An unsatisfiable subformula (a US) ofφ is a minimal-unsatisfiable subformula (MUS) if it becomes satisfiable whenever any of its clauses is removed. Algorithms for finding MUSes are presented in [1, 2, 7].

Since an unsatisfiable formula might have many MUSes, specific ones might be of greater value based on the application. For example, in verification applications the quality of refinement affects the number of iterations in the abstraction-refinement flow. While a US represents a set of spurious behaviors, an MUS represents a larger set of spurious behaviors. Thus, an MUS in general, and a smallest cardinality MUS (SMUS) in particular, tend to be more effective in reducing the number of refinement steps. Lynce et al. [4] presented an algorithm that computes an SMUS by implicitly searching all USes of a formula.

In this paper, we tackle the problem of finding an SMUS by a branch-and-bound algorithm that utilizes iterative MAXSAT solutions to generate lower and upper bounds on the size of the SMUS, and branch on specific subformulas to find it. The paper is

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 467 – 474, 2005.

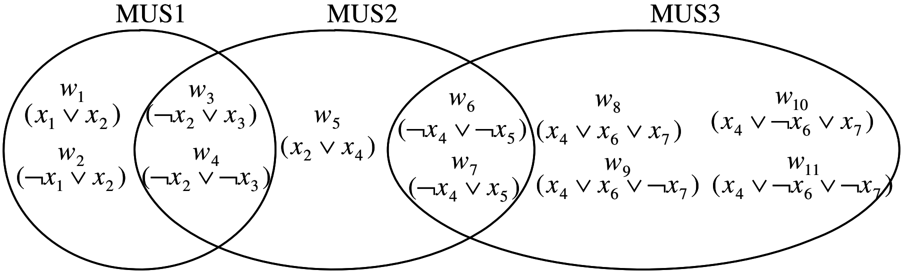

MUS1
MUS2
MUS3
w_1 $(x_1 \vee x_2)$
w_2 $(\neg x_1 \vee x_2)$
w_3 $(\neg x_2 \vee x_3)$
w_4 $(\neg x_2 \vee \neg x_3)$
w_5 $(x_2 \vee x_4)$
w_6 $(\neg x_4 \vee \neg x_5)$
w_7 $(\neg x_4 \vee x_5)$
w_8 $(x_4 \vee x_6 \vee x_7)$
w_9 $(x_4 \vee x_6 \vee \neg x_7)$
w_{10} $(x_4 \vee \neg x_6 \vee x_7)$
w_{11} $(x_4 \vee \neg x_6 \vee \neg x_7)$

Fig. 1. The formula $\prod$ of (1) and its MUSes

organized as follows. In Section 2, we review basic definitions and notations. In Section 3, we present our algorithm for finding the SMUS. Results on unsatisfiable formulas from DIMACS and DaimlerChrysler Automotive Product Configuration benchmarks are presented in Section 4.

2 Preliminaries

Consider an unsatisfiable formula $\varphi = w_1 w_2 \ldots w_m$. A US α ofφ is an MUS if removing every clause results in a formula that is satisfiable. α is an SMUS if it is an MUS and for all other MUSes β ofφ , $|\alpha| \leq |\beta|$. We denote the set of MUSes of φ by $Muses(\varphi)$. The MAX-SAT problem finds a satisfiable subformula α of φ with the maximum number of clauses; we call $\varphi - \alpha$ a MAX-SAT solution of φ. We solve MAX-SAT by reducing it to an integer optimization problem as follows. We define a set of m new Boolean *clause selector* variables $Y = \{y_1, y_2, \ldots, y_m\}$, and construct a new formula $\phi = (\neg y_1 \vee w_1)(\neg y_2 \vee w_2)\ldots(\neg y_m \vee w_m)$. The MAX-SAT solution is obtained by maximizing the objective $y_1 + y_2 + \ldots + y_m$ subject to the clauses of ϕ . Consider the Boolean formula:

$$\begin{aligned}\varphi = {} & (x_1 \vee x_2)(\neg x_1 \vee x_2)(\neg x_2 \vee x_3)(\neg x_2 \vee \neg x_3)(x_2 + x_4)(\neg x_4 \vee \neg x_5)(\neg x_4 \vee x_5) \\ & (x_4 \vee x_6 \vee x_7)(x_4 \vee x_6 \vee \neg x_7)(x_4 \vee \neg x_6 \vee x_7)(x_4 \vee \neg x_6 \vee \neg x_7)\end{aligned} \tag{1}$$

We call w_i $1 \leq i \leq 11$ the *ith* clause of φ.φ has three MUSes that are illustrated with a Venn diagram in Figure 1. MUS1 is an SMUS. There are several possible MAX-SAT solutions for φ. Two of them are $\{w_3, w_7\}$, and $\{w_1, w_6\}$.

3 Computing a Smallest MUS

A simple approach to compute an SMUS of a formula is to generate all MUSes and then select the smallest MUS. This approach is hindered by the fact that the number of MUSes of a formula can be exponential in the number of its variables. To solve this problem efficiently, we present a branch-and-bound algorithm to compute the SMUS.

3.1 Lower and Upper Bounds

We utilize iterative MAX-SAT solutions to get a lower bound on the size of the SMUS. Let us consider φ in (1) and its extended φ' with selector variables. We have seen that one possible MAX-SAT solution is $\{w_3, w_7\}$. From this, and the properties of MAX-SAT, we can conclude that every MUS of φ contains w_3 or w_7 (or both), and consequently contains at least one clause. To improve this lower bound, we repeat the above process by finding another MAX-SAT solution that contains clauses other than w_3 and w_7. This can be achieved by adding the constraints (y_3) and (y_7) to the MAX-SAT optimization problem to get: Maximize $\sum_{i=1}^{11} y_i$ subject to $(\phi)(y_3)(y_7)$. $\{w_4, w_6\}$ is a possible solution. Thus, every MUS must contain one of these clauses. In the third iteration, we have the following optimization problem: Maximize $\sum_{i=1}^{11} y_i$ subject to $(\phi)(y_3)(y_7)(y_4)(y_6)$ which has the solution $\{w_1, w_5, w_8\}$.

After adding the constraints (y_1), (y_5) and (y_8), the optimization problem becomes UNSAT because of the MUS $\{w_3, w_4, w_5, w_6, w_7\}$. The aggregated set of clauses from MAX-SAT solutions is $\varphi_{maxsat} = \{w_1, w_3, w_4, w_5, w_6, w_7, w_8\}$. At this point, we can conclude that any MUS contains at least 3 clauses since:

$$\begin{aligned}\forall \alpha \sqsubseteq \varphi \quad \alpha \text{ is MUS} \rightarrow \{w_3, w_4, w_1\} &\subseteq \alpha \vee \{w_3, w_4, w_5\} \subseteq \alpha \vee \\ \{w_3, w_4, w_8\} &\subseteq \alpha \vee \ldots \vee \{w_7, w_6, w_1\} \subseteq \alpha \vee \ldots \vee \\ \{w_7, w_6, w_8\} &\subseteq \alpha\end{aligned} \qquad (2)$$

Thus, the number of iterations of MAX-SAT is a lower bound (LB) on the number of clauses of the SMUS of φ.

To obtain an upper bound on the size of the SMUS, we can generate all the MUSes of the subformula φ_{maxsat}. The upper bound on the size of the SMUS is the size of the smallest MUS found in φ_{maxsat}. For our example φ_{maxsat} has a single MUS of size 5, and consequently the upper bound is 5.

3.2 Branch-and-Bound

Given $Muses(\varphi_{maxsat})$ and the initial LB and UB, if LB is equal to UB, then an MUS whose size is UB is an SMUS for φ. If this is not the case, we search the remaining MUSes of φ (the ones not in φ_{maxsat}) for an MUS (if any) whose size is smaller than UB. We achieve this by recursively branching on specific subformulas of φ, and bounding the search using LB and UB. The subformulas we branch on are $\varphi - \delta_i$ where $\delta_i \in \Delta$ and Δ is the set of all MAX-SAT solutions of φ_{maxsat}. Each of the subformulas in this recursion returns its SMUS if it is smaller than the one currently found in φ (and consequently φ's UB is updated). Otherwise, an empty set is returned. For the running example, all MAX-SAT solutions of φ_{maxsat} are: $\{w_3\}$, $\{w_4\}$, $\{w_5\}$, $\{w_6\}$ and $\{w_7\}$. Thus, five recursive calls are made on the subformulas $\varphi - \{w_3\}$, $\varphi - \{w_4\}$, $\varphi - \{w_5\}$, $\varphi - \{w_6\}$, and $\varphi - \{w_7\}$.

To understand why this approach efficiently searches the space of $Muses(\varphi) - Muses(\varphi_{maxsat})$, we have to address the following questions. Why will branching on all the specified subformulas “implicitly” search every MUS in $Muses(\varphi) - Muses(\varphi_{maxsat})$? How does a child subformula use its parent’s lower bound and MAX-SAT solution to compute its own lower bound? Finally, why is this an efficient solution? The first question addresses completeness, and its answer is omitted due to space limitations. We address the last two questions in what follows.

We use parent lower bound (pLB) and parent upper bound (pUB) to designate the upper and lower bounds of the parent formula, and current upper bound (cUB) and current lower bound (cLB) to designate the upper and lower bounds of a subformula $\varphi - \delta_i$. To understand how to compute $\varphi - \delta_i$’s cLB, let us consider the subformula $\varphi - \{w_3\}$. It is easy to verify that $Muses(\varphi - \{w_3\})$ includes all MUSes of φ except the ones that contain w_3. Since $Muses(\varphi - \{w_3\}) \subseteq Muses(\varphi)$, then MUSes of $\varphi - \{w_3\}$ satisfy (2), and pLB (that of φ) holds for $\varphi - \{w_3\}$. In fact, since all the MUSes of $\varphi - \{w_3\}$ do not contain w_3, they have to satisfy a stricter version of (2):

Algorithm 1 FindSMUS

```
FindSMUS($\varphi$)
    smus = FindSMUSRec($\varphi$, $\phi$ , 0, $\varphi \cdot size()$);
    if(smus == $\phi$) print "NO MUS. Formula is Satisfiable";
    else print smus;
FindSMUSRec(set $\varphi$, set pMaxSat, int pLB, int pUB)
    if(IsSat($\varphi$)) return $\phi$ ;
    (comp,numIter,cMaxSat)=IterateMaxSat($\varphi$,pMaxSat,pUB-pLB);
    if(!comp) return $\phi$;
    cLB = pLB + numIter;
      $\varphi_{maxsat}$ = cMaxSat + pMaxSat ;
    (muses, allMaxSats) = FindAllMuses($\varphi_{maxsat}$);
    cMUS = Smallest(muses);
    cUB = cMUS.size();
    smallestTillNow = (cUB < pUB)? cUB: pUB;
    set smallestMUS = (cUB < pUB)? cMUS: $\phi$;
    if(smallestTillNow<=cLB+1) return smallestMUS;
    foreach (ms in allMaxSats)
        $\varphi_{new}$ = $\varphi - ms$ ;
        $maxsat_{rec}$ = $\varphi_{maxsat} - ms$ ;
        recMUS=FindSMUSRec($\varphi_{new}$, $maxsat_{rec}$,cLB, smallestTillNow);
        if(recMUS!= $\phi$)
            smallestTillNow = recMus.size();
            smallestMUS = recMUS;
            if(smallestTillNow == cLB + 1) return smallestMUS;
    return smallestMUS;
```

Fig. 2. The algorithm for finding an SMUS

$$\forall \alpha \subseteq \varphi - \{w_3\} \quad \alpha \text{ is MUS} \rightarrow \{w_7, w_4, w_1\} \subseteq \alpha \vee \{w_7, w_4, w_5\} \subseteq \alpha \vee \dots \vee \{w_6, w_7, w_8\} \subseteq \alpha \tag{3}$$

We can continue the iterations of MAX-SAT on the formula $\varphi - \{w_3\}$ starting with the set $\varphi_{maxsat} - \{w_3\}$. In other words, the initial optimization problem for

$\varphi_{maxsat} - \{w_3\}$ is: Maximize $y_1 + y_2 + y_4 \ldots + y_{11}$ subject to $(\phi - \{(\neg y_3 \vee \neg x_2 \vee \neg x_3)\})(y_7)(y_4)(y_6)(y_1)(y_5)(y_8)$. A possible solution for this problem is $\{w_9\}$. By combining this with (3), we know that the current lower bound for $\varphi - \{w_3\}$ is four. The next solutions are $\{w_{10}\}$ and $\{w_{11}\}$ (at this point, the optimization problem is unsatisfiable). Following the above reasoning, we have cLB = pLB + numIter where numIter is the number of MAX-SAT iterations in the subformula. Thus, cLB for $\varphi - \{w_3\}$ is 6, and consequently, the smallest MUS in $\varphi - \{w_3\}$ has at least 6 clauses. In fact, $\varphi - \{w_3\}$ has a single MUS of size 6. As a result, $\varphi - \{w_3\}$ does not contain an MUS smaller than the best we have till now (5 clauses). The above conclusion can be reached with fewer computations by noting that numIter must be at most pUB - pLB. If the optimization problem remains satisfiable after pUB-pLB iterations, the current subformula does not contain an MUS smaller than pUB and the search is bound.

Let us consider the next recursive call on the subformula $\varphi - \{w_6\}$ and the MAX-SAT solution $\varphi_{maxsat} - \{w_6\}$. We know that pLB and pUB are 3 and 5 respectively. The optimization problem is: Maximize $y_1 + \ldots + y_5 + y_7 + \ldots + y_{11}$ subject to $(\phi - \{(\neg y_6 \vee \neg x_4 \vee \neg x_5)\})(y_3)(y_7)(y_4)(y_1)(y_5)(y_8)$. $\{w_2\}$ is a solution. At this point, the optimization problem is unsatisfiable; consequently cLB = 4. Next, we generate all MUSes of the current MAX-SAT solution: $\{w_1, w_3, w_4, w_5, w_7, w_2\}$. We get the MUS: $\{w_1, w_2, w_3, w_4\}$. Since the size of this MUS is equal to cLB, we have found the smallest MUS in $\varphi - \{w_6\}$. Since this MUS is smaller than cUB for φ, we update cUB to reflect the smallest MUS we have up to this point. The recursive calls on the remaining subformula can proceed with pLB = 3 and pUB = 4. No smaller MUS is found in these formulas. Thus the smallest MUS for φ is $\{w_1, w_2, w_3, w_4\}$.

An additional optimization can be applied to enhance the above algorithm. Consider φ again. From (2), we know that each MUS contains at least 3 clauses. If there is an MUS that contains exactly three clauses then it must be in φ_{maxsat}. Since we did not find an MUS of size 3 in φ_{maxsat} then we know that the SMUS of φ has size at least cLB + 1. Using this observation, and after returning from the branch of the subformula $\varphi - \{w_6\}$, and updating cUB of φ to 4, we conclude that we have found the SMUS since cUB = cLB + 1.

The pseudo code for algorithm that follows from the above description is illustrated in Figure 2. FindMusRec() is the recursive procedure for finding the SMUS. It takes as arguments the formula φ, the parent's MAX-SAT clauses pMaxSat, the parent lower bound pLB, and the parent upper bound pUB. If φ is satisfiable, it contains no MUS and the empty set is returned. Otherwise, the procedure calls IterateMaxSat() using the arguments φ, pMaxSat, and pUB-pLB. IterateMaxSat() returns three values. The Boolean variable comp is set to 0 if the optimization problem remains satisfiable after running pUB-pLB iterations, and is set to 1 otherwise. If comp is set to 1, numIter is the number of iterations, and is cMaxSat is the set current MAX-SAT clauses. If comp is 0 then the formula does not contain an MUS smaller than cUB and the empty set is returned. Otherwise, cLB is set to pLB + numIter, and all the MUSes and MAX-SAT solutions of φ_{maxsat} are computed. If the smallest of these MUSes is equal to cLB or cLB

+ 1, it is returned as the sMUS for φ. If this is not the case, we branch on all MAX-SAT solutions of φ_{maxsat} in a depth-first manner. After each branch terminates, we update the smallest MUS of φ and designate it as an SMUS if its size is equal to cLB + 1. After all recursive calls end, the SMUS is returned.

4 Experimental Results

To experimentally evaluate the effectiveness of our algorithm, we implemented it in C++ and used Satzoo [6] to solve MAX-SAT problems. For generating all MUSes we use the algorithms in [3]. All experiments were conducted on a 2 GHz Pentium 4 machine having 1 GB of RAM and running the Linux operating system.

Table 1 lists the results for representative aim benchmarks from the DIMACS set. The number of clauses of the SMUS range between 10% and 40% of the total number of clauses. The short run time is due to the fact that the total number of MUSes in these formulas is very small.

Table 2 presents the results for representative unsatisfiable formulas from the DaimlerChrysler Automotive Product Configuration Benchmarks [5]. To our knowledge, there exists no previous work that shows the sizes of the SMUSes for these benchmarks. The last column reports the number of MUSes obtained by running the algorithm in [3]. In some cases, the algorithm does not terminate and consequently either no information or a lower bound on the number of MUSes is provided. The total number of MUSes for these formulas ranges from 1 to more than a million. The size of the SMUS ranges from 0.1% to 5.5% (the average size is 1%) of the size of its formula. This shows that the SMUSes for these formulas are very small. Even in the cases where the number of MUSes is extremely large, our algorithm was able to efficiently find the SMUS. This shows the effectiveness of the implicit search utilized by the branch-and-bound process. For C202_FS_SZ_84, C202_FW_SZ_103, C210_FW_SZ_90, and C210_FW_SZ_91 the number of MUSes in φ_{maxat} was very large. To limit the run time, a cut-off of 500 seconds was used when generating all MUSes. The size column for these benchmarks has the format n1:n2 where n1 is the LB and n2 is the smallest MUS found before time-out. The difference between the best MUS found in the time limit and the lower bound for these formulas is 6, 8, 6, and 14 respectively. Thus, even

Table 1. Results on Representative Aim Benchmarks

Benchmark	Variables	Clauses	SMUS Size	Time (sec)
aim-50-1_6-no-2	50	80	32	0.08
aim-50-2_0-no-1	50	100	22	0.01
aim-100-1_6-no-1	100	160	47	0.14
aim-100-2_0-no-2	100	200	39	0.1
aim-200-1_6-no-1	200	320	55	0.36
aim-200-2_0-no-1	200	400	53	0.3

Table 2. Results on DaimlerChrysler Benchmarks

Benchmark	Variables	Clauses	SMUS Size	Time (sec)	# MUSes
C168_FW_SZ_107	1583	5939	47	546.54	NA
C168_FW_SZ_41	1583	4727	26	257.39	NA
C168_FW_SZ_66	1583	4751	16	18.84	NA
C168_FW_UT_2463	1804	6756	35	350.41	NA
C168_FW_UT_2469	1804	6767	32	831.46	NA
C168_FW_UT_714	1804	6754	9	14.49	NA
C168_FW_UT_851	1804	6758	8	59.91	102
C170_FR_RZ_32	1528	4067	227	121.33	32768
C170_FR_SZ_58	1528	4083	46	15.329	>16140
C170_FR_SZ_92	1528	4195	131	15.12	1
C170_FR_SZ_96	1528	4068	53	322.76	>172032
C202_FS_RZ_44	1556	5399	18	131.04	>79336
C202_FS_SZ_104	1556	5405	24	4.99	>1109330
C202_FS_SZ_121	1556	5387	22	2.5	4
C202_FS_SZ_122	1556	5385	33	3.84	1
C202_FS_SZ_74	1556	5561	150	36.43	NA
C202_FS_SZ_84	1556	5479	213:219	3878.3	NA
C202_FS_SZ_97	1556	5452	28	62.13	>63936
C202_FW_RZ_57	1561	7434	213	58.34	1
C202_FW_SZ_100	1561	7484	23	173.97	NA
C202_FW_SZ_103	1561	9024	147:155	8606.1	NA
C202_FW_SZ_123	1561	7437	36	14.74	4
C202_FW_SZ_61	1561	7490	18	163.84	NA
C202_FW_SZ_77	1561	7611	156	37.16	NA
C202_FW_SZ_98	1561	7438	7	58.16	NA
C208_FA_RZ_43	1516	4254	8	76.88	>9542
C208_FA_SZ_120	1516	4247	34	3.8	2
C208_FA_SZ_87	1516	4255	18	15.10	12884
C208_FA_UT_3254	1805	6153	40	95.27	17408
C208_FA_UT_3255	1805	6156	40	94.59	52736
C210_FS_RZ_23	1608	4911	31	266.30	NA
C210_FS_RZ_38	1607	4900	25	261.92	>188688
C210_FS_RZ_40	1607	4891	140	36.24	15
C210_FS_SZ_103	1607	4915	45	386.38	NA
C210_FS_SZ_107	1607	4902	15	25.29	NA
C210_FS_SZ_123	1607	5062	176	1401.97	>972463
C210_FS_SZ_78	1607	5071	170	56.72	NA
C210_FW_RZ_57	1628	6390	25	355.00	>129272
C210_FW_RZ_59	1628	6381	140	56.69	15
C210_FW_SZ_106	1628	6405	49	789.58	NA
C210_FW_SZ_111	1628	6393	15	35.17	NA
C210_FW_SZ_128	1628	6401	22	151.33	NA
C210_FW_SZ_90	1628	6977	271:277	6404.18	NA
C210_FW_SZ_91	1628	6709	267:281	6329.91	NA
C220_FV_RZ_12	1530	4017	11	50.58	>56872
C220_FV_RZ_13	1530	4014	10	33.35	6772
C220_FV_RZ_14	1530	4013	11	33.89	80
C220_FV_SZ_46	1530	4014	17	78.44	>5160
C220_FV_SZ_65	1530	4014	23	18.85	>84943

when the number of MUSes is very large, our algorithm provides useful information by generating an MUS whose size is close to the lower bound. We can see a large run time for these formulas. Most of this run time was spent computing φ_{maxsat}.

5 Conclusions

Understanding the causes of infeasibility of Boolean formulas is of interest in various theoretical and practical areas of computer science. Minimal unsatisfiable subformulas provide useful explanations of infeasibility. We have presented an algorithm to find an SMUS of a Boolean formula: an MUS with the least number of clauses. The algorithm utilizes the relation between MAX-SAT and MUSes to construct lower and upper bounds on the size of the SMUS. These bounds are the basis for a branch-and-bound procedure that finds the SMUS by recursively branching on specific subformulas. We have presented novel experimental results on two benchmark suites.

Acknowledgment

This work was funded in part by the National Science Foundation under ITR grant number No. 0205288.

References

[1] R. Bruni and A. Sassano, “Restoring Satisfiability or Maintaining Unsatisfiability by finding *small* Unsatisfiable Subformulae,” in *Electronic Notes in Discrete Mathematics*, vol. 9, 2001.

[2] J. Huang, “MUP: A Minimal Unsatisfiability Prover,” in Proceedings of Asia South Pacific Design Automation Conference, 2005.

[3] M. Liffiton, Z. Andraus, and K. Sakallah, “From MAX-SAT to Min-UNSAT: Insights and Applications,” Technical Report CSE-TR-506-05, University of Michigan, 2005.

[4] I. Lynce and J. Marques-Silva, “On Computing Minimum Unsatisfiable Cores”, in *Seventh International Conference on Theory and Applications of Satisfiability Testing* (SAT), 2004.

[5] SAT benchmarks from Automotive Product Configuration,
http://www-sr.informatik.unituebingen.de/~sinz/DC/

[6] Satzoo, http://www.cs.chalmers.se/~een/Satzoo/

[7] L. Zhang and S. Malik, “Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula,” in Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), S. Margherita Ligure - Portofino, Italy, 2003.

Threshold Behaviour of WalkSAT and Focused Metropolis Search on Random 3-Satisfiability

Sakari Seitz[1,2], Mikko Alava[2], and Pekka Orponen[1]

[1] Laboratory for Theoret. Computer Science, P.O.B. 5400
[2] Laboratory of Physics, P.O.B. 1100,
Helsinki University of Technology,
FI-02015 TKK, Finland
firstname.lastname@tkk.fi

Abstract. An important heuristic in local search algorithms for Satisfiability is *focusing*, i.e. restricting the selection of flipped variables to those appearing in presently unsatisfied clauses. We consider the behaviour on large randomly generated 3-SAT instances of two focused solution methods: WalkSAT and Focused Metropolis Search. The algorithms turn out to have qualitatively quite similar behaviour. Both are sensitive to the proper choice of their “noise” and “temperature” parameters, but with appropriately chosen values, both achieve solution times that scale linearly in the number of variables even for clauses-to-variables ratios $\alpha > 4.2$. This is much closer to the satisfiability transition threshold $\alpha_c \approx 4.267$ than has generally been assumed possible for local search algorithms.

1 Introduction

It was observed in [10] that random 3-SAT instances change from being generically satisfiable to being generically unsatisfiable when the clauses-to-variables ratio $\alpha = M/N$ exceeds a critical threshold α_c. Current numerical and analytical estimates [3] suggest that this satisfiability threshold is located approximately at $\alpha_c \approx 4.267$. It was also suggested in [10] that the satisfiability of random 3-SAT formulas is hardest to decide in the region close to α_c.

The space of solutions to 3-SAT instances slightly below α_c develops structure: at $\alpha \approx 3.92$ the solutions become “clustered” [3], with solutions belonging to the same cluster being geometrically close to each other. Such clusters may possess extensive “backbones”: sets of variables whose values are fixed across the cluster, while the values of others can be varied subject to some constraints.

Good results have recently been reported on finding satisfying assignments to large random formulas at $\alpha \lesssim \alpha_c$ using the *survey propagation* method [3], which iterates a guess about the state of each variable, in the course of fixing an ever larger proportion of the variables to their “best guess” values. In this paper we argue that also simple local search methods, i.e. algorithms that try to

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 475–481, 2005.

find a solution by flipping the value of one variable at a time, can achieve comparable performance levels. (There is also some evidence [7] that local search methods may be more robust than survey propagation on structured SAT instances.)

When the variables to be flipped are chosen only from the unsatisfied clauses, a local search algorithm is called *focusing.* A well-known focused method for 3-SAT is WalkSAT [17], which interleaves random variable flips among greedy ones according to a given "noise" probability p. We shall contrast WalkSAT with a focused variant of the standard Metropolis dynamics [9], which we call in this setting Focused Metropolis Search (FMS).

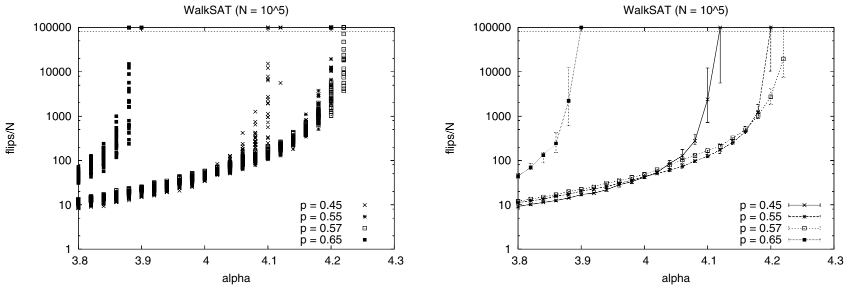
We demonstrate that both WalkSAT and FMS work in the "critical" region up to $\alpha > 4.2$, given appropriate parameter choices. For instance, the solution times achieved by FMS are found to be linear in N within a parameter window $\eta_{min} \leq \eta \leq \eta_{max}$, where η is essentially the Metropolis temperature. Within this window, the median and other quantiles of the solution time per variable approach a constant value as N increases. For too large η the algorithm becomes "entropic", while for too small η it is too greedy, leading to freezing of degrees of freedom. For increasing α, the optimal $\eta = \eta_{opt}(\alpha)$ increases and the linearity window gets narrower. Based on these characteristics, we postulate a phase diagram for FMS, with two phase boundaries resulting from too little or too much greed in the choice of η.

An extended version of this paper, available as a technical report [16], contains some further experiments covering also the so called Focused Record-to-Record Travel (FRRT) algorithm [4, 15], and a discussion about how *whitening* [12] relates to the dynamics of focused local search.

2 Local Search for Satisfiability

It is natural to view the satisfiability problem as a combinatorial optimisation task, where the goal for a given formula F is to minimise the "energy" function $E = E_F(s)$ = the number of unsatisfied clauses in formula F under truth assignment s. The successful WalkSAT algorithm [17] combines the technique of focusing, discussed earlier by Papadimitriou [11] in the context of a purely stochastic Random Walk algorithm, with interleaved greedy and random steps.

Several studies on the efficiency of different variants of WalkSAT have been published (e.g. [5, 6, 8, 13, 20]). Recently, Barthel et al. [2] performed systematic numerical experiments with Papadimitriou's original Random Walk method at $N = 50,000$, $\alpha = 2.0 \ldots 4.0$. They also explained a transition in the dynamics of this algorithm at $\alpha_{\mathrm{dyn}} \approx 2.7$, observed earlier also in [14]. When $\alpha < \alpha_{\mathrm{dyn}}$, satisfying assignments are generically found in time that is linear in the number of variables, whereas when $\alpha > \alpha_{\mathrm{dyn}}$ exponential time is required. (Similar results were obtained by Semerjian and Monasson in [18], though with smaller experiments ($N = 500$).) At $\alpha > \alpha_{\mathrm{dyn}}$ the search equilibrates at a nonzero energy level, and only escapes to a ground state through a large enough ran-



(a) Complete data

(b) Medians and quartiles

Fig. 1. Normalised solution times for WalkSAT at $\alpha = 3.8 \ldots 4.3$, $N = 10^5$

dom fluctuation. A rate-equation analysis of the method [2] yields an approximation $\alpha_{\text{dyn}} \approx 2.71$.[1] See also [19] for further analyses of the Random Walk method.

WalkSAT is more powerful than the simple Random Walk, because in it focusing is accompanied by other heuristics. It is known that the behaviour of WalkSAT is quite sensitive to the choice of the noise parameter p, and that the optimal choice of p depends on the particulars of the problem at hand and the variant of WalkSAT used [6, 5, 8, 20]. Concerning the ideal behaviour of the algorithm on random 3-SAT instances, Parkes [13] experimented with a noise value $p = 0.3$ and concluded that the algorithm works in linear time at least up to $\alpha = 3.8$. In other sources [5, 6, 20] it has been estimated that close to the satisfiability transition the optimal noise setting is $p \approx 0.55$.

Nevertheless, it has been conjectured (e.g. in [3]) that no local search algorithm can work in linear time beyond the clustering transition at $\alpha \approx 3.92$. In a series of recent experiments, however, Aurell et al. [1] concluded that with a proper choice of parameters, the median solution time of WalkSAT remains linear in N up to at least $\alpha = 4.15$, the onset of "1-RSB symmetry breaking". Our experiments indicate that the median time in fact remains linear even beyond that, in line with the results presented in [15].

Figure 1 illustrates our experiments with WalkSAT[2] on randomly generated formulas of size $N = 10^5$, various values of the noise parameter p, and $\alpha = 3.8 \ldots 4.22$. For each (p, α)-combination, 21 formulas were generated, and for each of these the algorithm was run until either a satisfying solution was found or a time limit of $80000 \times N$ flips was exceeded. Figure 1(a) shows the solution times t_{sol}, measured in number of flips normalised by N, for each generated formula. Figure 1(b) gives the medians and quartiles of the data

[1] Our numerical experiments actually suggest a somewhat lower value, $\alpha_{\text{dyn}} \approx 2.67$.

[2] Version 43, downloaded from `http://www.cs.washington.edu/homes/kautz/walksat/`, with its default heuristics.

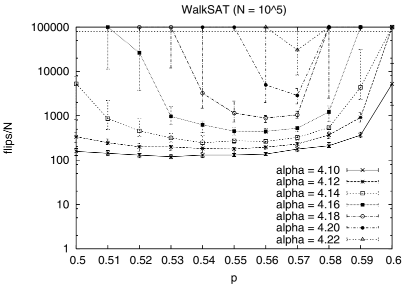
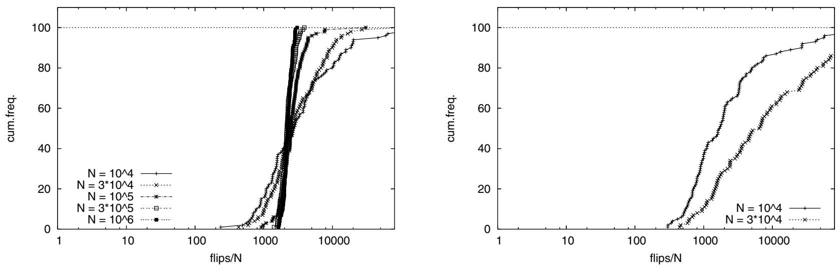


Fig. 2. Normalised solution times for WalkSAT with $\alpha = 4.10 \ldots 4.22$, $p = 0.50 \ldots 0.60$

for each value of α. As can be seen from the figures, for value $p = 0.45$ of the noise parameter WalkSAT finds satisfying assignments in roughly time linear in N, with the coefficient of linearity increasing gradually with increasing α, up to approximately $\alpha = 4.1$ beyond which the distribution of solution times for the algorithm diverges. For $p = 0.55$, this linear regime extends further, up to at least $\alpha = 4.18$, but for $p = 0.65$ it seems to end already before $\alpha = 3.9$. For the best value of p we have been able to experimentally determine, $p = 0.57$, the linear regime seems to extend up to even beyond $\alpha = 4.2$.

In order to estimate the value of p for which the linear time regime extends to largest values of α, we generated test sets consisting of 21 formulas, each of size $N = 10^5$, for α values ranging from 4.10 to 4.22, and for each α for p values ranging from 0.50 to 0.60. WalkSAT was then run on each resulting (α, p) test set; the medians and quartiles of the observed solution time distributions are shown in Figure 2. The data suggest that the asymptotically optimal value of the noise parameter is $p \approx 0.57$. Figure 3 shows the empirical solution time distributions over 100 runs at $\alpha = 4.20$ for $p = 0.57$ and $p = 0.55$. For $p = 0.57$ the quantiles seem to converge to a finite value for increasing N, but for $p = 0.55$, the distributions seem to diverge, with median values increasing with N.



(a) $\alpha = 4.15$

(b) $\alpha = 4.20$

Fig. 3. Solution time distributions for WalkSAT at $\alpha = 4.20$, $p = 0.57$ and $p = 0.55$

3 Focused Metropolis Search

From an analytical point of view, the WalkSAT algorithm is rather complicated. Thus, it is of interest to investigate the behaviour of the simpler algorithm obtained by imposing the focusing heuristic on a basic Metropolis dynamics [9]. We call the resulting algorithm the *Focused Metropolis Search* (FMS) method. The algorithm is parameterised by a number η, $0 \leq \eta \leq 1$, such that a candidate variable flip leading to an increase of $\varDelta E > 0$ in energy is accepted with probability $\eta^{\varDelta E}$. Thus in customary Metropolis dynamics terms, $\eta = e^{-1/T}$, where T is the chosen computational temperature. (Note, however, that detailed balance does not hold with focusing.)

The behaviour of the algorithm is qualitatively quite similar to WalkSAT. For $\eta = 0.2$, the linear time regime seems to extend up to roughly $\alpha = 4.06$, for $\eta = 0.3$ up to at least $\alpha = 4.14$; however for $\eta = 0.4$ again only up to maybe $\alpha = 4.08$. To determine the optimal value of the η parameter we again mapped out systematically the solution time distributions of the FMS algorithm for α increasing from 4.10 to 4.22 and η ranging from 0.28 to 0.38. The results, shown in Figure 4, suggest that the optimal value of the parameter is approximately $\eta = 0.36$.

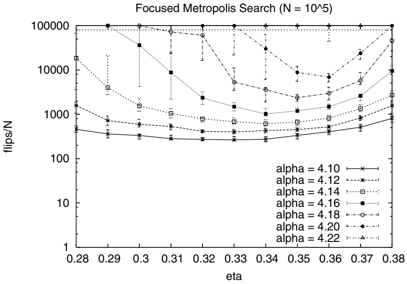


Fig. 4. Normalised solution times for FMS with $\eta = 0.28 \ldots 0.38$, $\alpha = 4.10 \ldots 4.22$

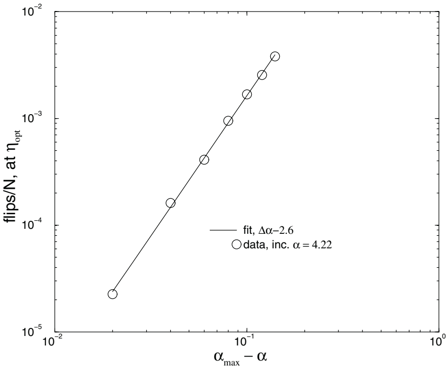


Fig. 5. Solution time t_{sol} at $\eta = \eta_{opt}(\alpha)$ for $N = 10^5$: poss. divergence at $\alpha_{max} = 4.24$

Using the data for varying η allows one to extract the optimal $\eta_{opt}(\alpha)$. The optimal η increases up to about $\alpha = 4.22 \ll \alpha_c$ roughly linearly. The corresponding optimal solution time at η_{opt} diverges, and our attempts to extract a *maximal* value α_{max} beyond which linearity no longer exists are inconclusive for lack of data. See Figure 5 for an example of a fit of the kind $t_{sol} \sim (\alpha_{max} - \alpha)^{-b}$.

4 Conclusions

Based on the previous experimental data, we propose in Figure 6 a phase diagram for FMS. Note that at $\eta = 1$ FMS coincides with the Random Walk algorithm; hence the first transition point is at $\alpha \approx 2.67$. For larger α there are *two phase boundaries* in terms of η. η_u separates the linear regime from one with too much noise, in which the fluctuations degrade algoritm performance. For $\eta < \eta_l$, greediness leads to dynamical freezing, so that the algorithm no longer scales linearly in N. Two questions then arise: what is the smallest α at which $\eta_l(\alpha)$ starts to deviate from zero; and how close to α_c can one push the linear regime by an appropriate choice of η_{opt}? The resolution of these issues will depend on our understanding of local search methods in the presence of the clustering of solutions. The clustered solutions should have an extensive core of frozen variables, and therefore be hard to find. Since FMS performs well even in this case, not all relevant features of the solution space and energy landscape are presently understood.

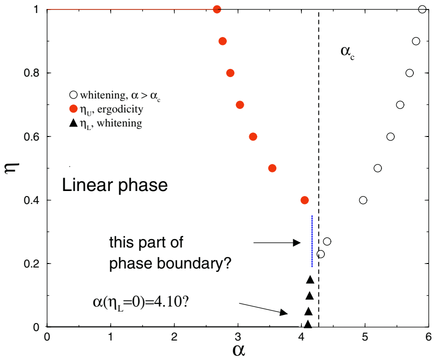


Fig. 6. Proposed phase diagram for FMS. Above a dynamical threshold a metastable state is encountered before a solution is reached. Below a "whitening" threshold a freezing transition happens before a solution or a metastable or stable state is reached

Acknowledgements. We are most grateful to Dr. Supriya Krishnamurthy for useful comments and discussions. This research was supported by the Academy of Finland, grants 206235 (S. Seitz) and 204156 (P. Orponen), and by the Center of Excellence program (M. Alava).

References

1. E. Aurell, U. Gordon, S. Kirkpatrick, Comparing beliefs, surveys and random walks. *NIPS 2004*. arXiv.org: cond-mat/0406217.
2. W. Barthel, A. K. Hartmann, M. Weigt, Solving satisfiability problems by fluctuations: The dynamics of stochastic local search algorithms. *Phys. Rev. E 67* (2003), 066104.
3. A. Braunstein, M. Mézard, R. Zecchina, Survey propagation: an algorithm for satisfiability. arXiv.org:cs.CC/0212002.
4. G. Dueck, New optimization heuristics: the great deluge algorithm and the record-to-record travel. *J. Comput. Phys. 104* (1993), 86–92.
5. H. H. Hoos, An adaptive noise mechanism for WalkSAT. *AAAI 2002*, 655–660.
6. H. H. Hoos, T. Stützle, Towards a characterisation of the behaviour of stochastic local search algorithms for SAT. *Artif. Intell. 112* (1999), 213–232.
7. H. Jia, C. Moore, B. Selman, From spin glasses to hard satisfiable formulas. *SAT'04*.
8. D. McAllester, B. Selman, H. Kautz, Evidence for invariants in local search. *AAAI 1997*, 321–326.
9. N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller, Equations of state calculations by fast computing machines, *J. Chem. Phys. 21* (1953), 1087–1092.
10. D. Mitchell, B. Selman, H. Levesque, Hard and easy distributions of SAT problems. *AAAI 1992*, 459–465.
11. C. Papadimitriou, On selecting a satisfying truth assignment. *FOCS 1991*, 163–169.
12. G. Parisi, On local equilibrium equations for clustering states. arXiv.org:cs.CC/0212047.
13. A. J. Parkes, Distributed local search, phase transitions, and polylog time. *Proc. Workshop on Stochastic Search Algorithms, IJCAI 2001.*
14. A. J. Parkes, Scaling properties of pure random walk on random 3-SAT. *CP 2002*, 708–713. Springer LNCS 2470.
15. S. Seitz, P. Orponen, An efficient local search method for random 3-satisfiability. *Workshop on Typical Case Complexity and Phase Transitions, LICS 2003.* Electronic Notes in Discrete Mathematics 16, Elsevier 2003.
16. S. Seitz, M. Alava, P. Orponen, Focused local search for random 3-satisfiability. arXiv.org:cond-mat/0501707.
17. B. Selman, H. Kautz, B. Cohen, Local search strategies for satisfiability testing. *Cliques, Coloring, and Satisfiability*, 521–532. AMS DIMACS Series 26, 1996.
18. G. Semerjian, R. Monasson, Relaxation and metastability in a local search procedure for the random satisfiability problem. *Phys. Rev. E 67* (2003), 066103.
19. G. Semerjian, R. Monasson, A study of Pure Random Walk on random satisfiability problems with "physical" methods. *SAT 2003*, 120–134. Springer LNCS 2919.
20. W. Wei, B. Selman, Accelerating random walks. *CP 2002*, 216–232. Springer LNCS 2470.

On Subsumption Removal and On-the-Fly CNF Simplification

Lintao Zhang

Microsoft Research Silicon Valley Lab,
1065 La Avenida Ave., Sunnyvale, CA 94043
lintaoz@microsoft.com

Abstract. CNF Boolean formulas generated from resolution or solution enumeration often have much redundancy. Efficient algorithms are needed to simplify and compact such CNF formulas. In this paper, we present a novel algorithm to maintain a subsumption-free CNF clause database by efficiently detecting and removing subsumption as the clauses are being added. We then present an algorithm that compact CNF formula further by applying resolutions to make it *Decremental Resolution Free*. Our experimental evaluations show that these algorithms are efficient and effective in practice.

1 Introduction and Previous Works

Let *l(C)* denotes the set of literals of a clause *C* in a CNF Boolean formula. Given clauses C_1 and C_2, if $l(C_1) \in l(C_2)$, then C_1 subsumes C_2 and C_2 is subsumed by C_1. A subsumed clause is redundant and can be removed from the formula without changing the Boolean function it represents. Since redundant clauses consume memory and slow down reasoning, it is desirable to make a CNF formula subsumption free by detecting and removing subsumed clauses. Modern SAT solvers and QBF solvers usually maintains a CNF database, which is modified during the solving process due to mechanisms such as learning (e.g. [1] [2]) and resolution and expansion (e.g. [3][4]). It is desirable if subsumption removal can be performed on-the-fly.

When a new clause is added to a formula, existing clauses in the database are checked against it to see whether they are subsumed. This check is called *backward subsumption* checking. The new clause is also checked against existing clauses to see if it is subsumed by any of them. This check is called *forward subsumption* checking. To keep the clause database subsumption-free, both checks need to be performed.

Efficient implementation of subsumption removal and, in a broader sense, on-the-fly CNF simplification, haven't received enough attention in SAT research. In search based solvers, learned clauses can be deleted if necessary. Therefore, memory and run time overhead for the subsumed clauses can be kept under control. Recently new SAT and QBF algorithms and applications make subsumption removal more relevant. Some recent solvers (e.g. Quantor [4]) and preprocessor (e.g. NiVer [5]) are rekindling the use of resolution. SAT based solution enumeration (e.g. [6]), which tries to enumerate all the solutions of a SAT instance, is attracting a lot of attention in the

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 482 – 489, 2005.

EDA community. In both of these cases, newly added clauses cannot be removed if they are not redundant. Therefore, subsumed clauses must be detected and removed in order to free the memory and increase the capacity of the solver.

de Kleer [7] proposed to use trie data structure to detect and remove subsumption. Chatalic and Simon [8] proposed a subsumption elimination operator and a subsumption-free union operator on formulas represented as Zero-suppressed Binary Decision Diagrams (ZBDDs [9]) to make the CNF subsumption free. Both trie and ZBDD incur significant overhead for maintaining the clause database. They are not widely used in modern solvers. Biere [4] described a signature based backward subsumption detection algorithm for flat CNF databases. Each time a clause is added to the database, a signature is computed and checked against existing clauses. Signatures incur extra memory overhead. Moreover, In [4] the author did not describe a forward subsumption detection algorithm. In fact, Forward subsumption was detected by periodically removing all clauses, flushing signatures and adding back the clauses in reverse chronological order.

The problem of subsumption detection and removal in CNF can be regarded as a restricted case of 2-level logic simplification. It's well known that prime and irredundant covers for a Boolean function can be computed using e.g. iterative consensus and min-cover algorithms [10]. Some authors (e.g. [11]) attempted to use existing logic simplification techniques for CNF simplification. Unfortunately, the algorithms used for logic simplification are usually too expensive for CNF formulas, which often involve thousands of variables and clauses. Moreover, such techniques cannot be applied on-the-fly. In [6], the authors described a technique to perform on-the-fly clause compaction and simplification for SAT based solution enumeration and quantification. The algorithm can only detect possible simplifications between two clauses with same number of literals. It does not guarantee the CNF to be subsumption free.

In this paper, we propose an efficient algorithm for on-the-fly subsumption removal for CNF formulas. We propose an alternative backward subsumption detection algorithm that does not incur memory overhead as [4] does. We also propose an efficient forward subsumption detection algorithm inspired by the two-literal-watching algorithm of Chaff [12]. Unlike [7] and [8], our algorithms operate on a flat CNF clause database. To compact the CNF database even more. We propose an algorithm that recursively applies the combination of a slightly modified version of afore mentioned algorithms and resolution. The algorithm not only makes the clause database subsumption free, but also *decremental resolution free*. In contrast to [6], our algorithm can detect possible simplification between two clauses that have unequal number of literals. Moreover, it still maintains the subsumption-freeness of the CNF.

2 Subsumption Detection and Removal Algorithms

2.1 Backward Subsumption Detection

Given a clause C, the backward subsumption detection algorithm finds all clauses whose literals are supersets of the literals in C. For each literal l, we keep a list of clauses in which this literal occurs, denote as *ClauseSet(l)*. The pseudo code for backward subsumption detection is shown in the Algorithm 1.

This algorithm can be implemented efficiently. Set intersection has a linear run time complexity to the total number of elements in the sets. Therefore, the worst case complexity for finding clauses backward subsumed by clause *C* is linear to the total number of clauses in which the literals in *C* occur. In practice, the literals in a clause are sorted in ascending order with regard to the number of clauses they appear in. The iteration often ends with just a couple of iterations when S becomes empty.

The advantage of this algorithm compared with the signature base algorithm in [4] is that it does not incur any overhead for storing signatures. Note that the set of clauses a literal occurs has to be kept by both methods. In fact, this data structure is often maintained by the existing SAT solvers or preprocessors for various other reasons. Therefore, it can be regarded as free.

```
BackwardSubsumedBy ( C )
{
  S = Set of All the Clauses;
  For each literal l in C {
    S = S ∩ ClauseSet ( l );
    If S is empty then break;
  }
  return S;
}
```

Algorithm 1. Backward Subsumption Detection

```
IsForwardSubsumed ( C )
{
  for each literal l in C {
    mark l;
    for each clause $C_1$ in WCls(l)
{
      $l_1$ = a literal in $C_1$
           that is not marked;
      if (no such $l_1$) {
        unmark all literals in C;
        return true;
      }
      else {
        remove $C_1$ from WCls (l);
        put $C_1$ in WCls ($l_1$);
      }
    }
  }
  unmark all literals in C;
  return false;
}
```

Algorithm 2. Forward Subsumption Detection

```
AddClauseAndMaintainDRF ( C )
{
  sub_cls = clauses that contain subsets
            of variables in C;
  d0_sub  = clauses in sub_cls that are
            distance 0 from C;
  if (d0_sub is not empty) //C is sub-
sumed
    return;
  d1_sub  = clauses in sub_cls that are
            distance 1 from C;
  if ( d1_sub is not empty ) {
    $C_1$ = choose a clause in d1_sub;
    $C_0$ = resolvant of C and $C_1$;
    AddClauseAndMaintainDRF ( $C_0$ );
    return;
  }
  sup_cls = clauses that contain super-
sets
            of variables in C;
  d0_sup  = clauses in sup_cls that are
            distance 0 from C;
  d1_sup  = clauses in sup_cls that are
            distance 1 to C;
  remove all clauses in d0_sup from CNF;
  if ( d1_sup is not empty ) {
    for each $C_1$ in d1_sup {
      remove $C_1$ from CNF;
      $C_0$ = resolvant of C and $C_1$;
      AddClauseAndMaintainDRF ( $C_0$ );
    }
    AddClauseAndMaintainDRF ( C );
    return;
  }
  add clause C to CNF;
}
```

Algorithm 3. Maintaining a DRF Clause Database

2.2 Forward Subsumption Detection

Given a clause *C*, forward subsumption detection algorithm finds out if any clause in the current database consists of a subset of the literals in *C*. If such a clause exists, then clause *C* is subsumed by it and is redundant. The algorithm for forward subsumption detection is based on the fact that if a clause *C* is a conflicting clause, then all the clauses that subsume it must also be conflicting clauses. Assume we set all literals in C to be *false*. If clause C_1 subsumes C, then all literals in C_1 must also be *false*.

Our algorithm to detect whether such C_1 exists is inspired by the 2-literal watching algorithm described in [12]. To detect if a clause is conflicting under a set of variable assignment, we only need to detect whether it contains at least one literal that is not false. We call this algorithm one-literal watching algorithm. Each clause has one literal marked as being watched. Each literal l has a list containing the set of clauses with watched literals corresponding to it. We denote the list as *WCls (l)*. Given a clause C, the algorithm *IsForwardSubsumed(C)* returns *true* or *false* depending on whether the clause C is forward subsumed. The complexity of forward subsumption detection for a clause C is about the same as applying n variable assignments (i.e. implications) on the CNF database for a SAT solver, where n is the number of literals in clause C.

A clause C is forward subsumed if an existing clause in the CNF subsumes C. If C is forward subsumed, then it is redundant and should not be added to the database. However, even if C is not subsumed, it may still be redundant. A clause is redundant if current CNF implies it. It is easy to prove that given Boolean formulas f_1 and f_2, if $\neg f_1 \wedge f_2$ is false, then f_2 implies f_1. Subsumption is a specific case of redundancy. We can easily devise other methods to detect redundancy. The following algorithms can be regarded as strengthened versions of the forward subsumption algorithm described previously.

A simple way to strengthen the forward subsumption detection algorithm is to apply full Boolean Constraint Propagation (BCP) when setting literals in C to be false. If it leads to a conflict, then C is redundant and should not be added to the CNF. Notice that in this case C may not be subsumed by any existing clauses. The complexity of this strengthened forward subsumption detection algorithm is about the same as applying n variable branches (i.e. decisions) on the CNF database, where n is the number of literals in C. It is obviously more costly than simple forward subsumption detection, but potentially more effective.

We can strengthen the algorithm even more. If after setting all literals in C to *false* and applying BCP do not produce a conflicting clause, we can apply a SAT solver on the resulting CNF to see if it is satisfiable. If it is unsatisfiable, then C is redundant and should not be added to the CNF. Obviously, the complexity of this version of redundancy detection is in the same order of a SAT solving, which can be very expensive. We can set a time limit on the solver, abort solving when time out, and conservatively assume the clause to be irredundant.

2.3 Maintaining a Subsumption Free CNF Database

By combining the forward and backward subsumption detection algorithms, we obtain the algorithm for maintaining a subsumption free CNF database. As clause enters the database, subsumed clauses are being detected and removed. Our experiments show that forward subsumption detection is much cheaper than backward subsumption detection. Therefore, when a clause enters the database, we first call procedure *IsForwardSubsumed*. If the clause is subsumed by existing clauses in the database, the clause is discarded and no further check is necessary. Otherwise, *BackwardSub-*

sumption is called to check if it subsumes any existing clauses. If so, the subsumed clauses are deleted and the new clause is added to the CNF.

3 Maintaining a Decremental Resolution Free CNF Database

A subsumption free CNF can often be simplified further by a resolution followed by subsumption. Consider clauses (a ∨ b) and (a ∨ ¬b ∨ c). It is easy to observe that by resolving these two clauses, the resolvant (a ∨ c) subsumes the second clause, therefore, the formula can be simplified as (a ∨ b) ∧ (a ∨ c). We will call a resolution that generates a clause that subsumes at least one of its parent clauses a *decremental* resolution. Formally, we define the *distance* between two clauses C_1 and C_2 as the number of variables that occur in both C_1 and C_2 but are in different polarities. Given clauses C_1 and C_2 with distance 1, the *resolvant* is the clause that contains all literals in C_1 and C_2 except the distance 1 literals. If the set of variables in C_1 is a subset of the variables in C_2, then such a resolution is a *decremental resolution* because the resolvant subsumes C_2. Given a CNF database, if no decremental resolution is possible between any pair of clauses, then the CNF is *decremental resolution free (DRF).* In the following, we assume that subsumption-freeness is always maintained. Therefore, when we say a CNF is decremental resolution free, we imply that it is also subsumption free.

Notice that an algorithm that maintains a decremental resolution free database may not necessarily generate a more compact representation than the algorithm that maintains a subsumption free database due to its greedy nature. Suppose we add three clauses (a ∨ ¬d), (a ∨ b ∨ c ∨ d), and (c ∨ d) into the database, in that order. If we maintain a subsumption free database, since the second clause is subsumed by the third one, we will remove it and obtain a CNF with 2 clauses and 4 literals. On the other hand, if we maintain a DRF database, the second clause resolves with the first clause resulting in clause (a ∨ b ∨ c), which is not subsumed by the third clause. Therefore, the end result is a CNF formula with 3 clauses and 7 literals.

The algorithm to maintain a DRF CNF is built upon algorithms for subsumption detection. In subsumption detection, we need to find clauses that contain subsets (as in forward subsumption) or supersets (as in backward subsumption) of the *literals* in a clause *C*. We can modify both of the algorithms slightly to find clauses that contain subsets or supersets of the *variables* in a clause *C*. This can be achieved easily. E.g. in the backward subsumption algorithm, we keep a list of clauses for each variable instead of each literals as *ClauseSet(v).* In forward subsumption algorithm, we set marks on variables instead of literals. The pseudo code for adding a clause into a DRF clause database is shown in Algoirthm 4.

Algorithm *AddClauseAndMaintainDRF* adds a clause into a CNF formula if and only if it can make sure that there is no possible subsumption or decremental resolution between the clause to be added and the clauses in current CNF. If subsumption or decremental resolution is possible, it eliminates them and then makes a recursive call to perform the check against the new CNF. One interesting implementation detail that needs to be pointed out is that after `d1_sup` is calculated, some of the clauses in it may be deleted by subsequent recursive calls. Therefore, whenever a clause from `d1_sup` is accessed in the `foreach` loop, it must be checked to make sure that the clause is still valid in current CNF.

4 Experimental Results

In this section, we report some preliminary experimental results to show the feasibility and effectiveness of our CNF simplification algorithms. Notice that the purpose of the experiment is not to demonstrate the usefulness of subsumption removal because that is very application specific. Therefore, we are not trying to answer questions like "How many clauses can be subsumed". "Is it worth the effort?" What we are demonstrating here is that for the set of algorithms we proposed in the paper that operates with little overhead, we can implement them reasonably efficiently for most applications to use. We are not comparing our algorithm with existing algorithms because none of the other algorithms operates under the same setting (i.e. support on-the-fly forward and backward subsumption detection on flat CNF databases).

Table 1. Variable Elimination by Resoulution: No CNF Simplification

Formula	Num. Vars Eliminated	Original Formula			After Resolution		Time (s)
		Num Vars	Num Cls	Num Lits	Num Cls	Num Lits	
3SAT_100_400	10	100	400	583	19333	180120	0.453
1dlx_c_mc_ex_bp_f	40	776	3725	10045	154180	2854150	4.641
c7552	700	7652	20423	46377	247714	3702725	10.172
bw_large.d	100	6325	131973	294118	212530	1102847	3.641
longmult15	250	7807	24351	58557	692817	4728551	12.656

Table 2. Variable Elimination: Applying Subsumption Removal

Formula	Total Added		Forward		Backward		Final Result		Time (s)
	Num Cls	Num Lits	Num Cls	Num Lits	Num Cls	Num Lits	Num Cls	Num Lits	
3SAT_100_400	3782	25919	2028	16489	141	915	1430	7810	0.203
1dlx_c_mc_ex_bp_f	69686	1073171	39240	706487	5406	83234	22836	260403	9.719
c7552	49259	2352969	3157	144807	869	4238	35316	1368639	18.406
bw_large.d	201832	919829	28673	436138	1462	5792	168678	469222	7.703
longmult15	121059	500805	60649	304472	1061	3425	57353	186920	5.047

Table 3. Variable Elimination: Maintaining DRF

Formula	Total Added		Final Result		Time (s)
	Num Cls	Num Lits	Num Cls	Num Lits	
3SAT_100_400	3007	17578	1235	5933	0.165
1dlx_c_mc_ex_bp_f	60665	817435	20967	211399	10.438
c7552	48708	2328013	35276	1355117	28.547
bw_large.d	189187	715388	166021	441140	12.422
longmult15	119450	445434	54761	160775	6.875

For this purpose we need to construct a set of input clauses to feed to the algorithms. The simple application we choose to test the CNF simplification techniques is as follows. We randomly choose a subset of the variables in a CNF formula and eliminate them one by one using resolution (i.e. Davis Putnam algorithm [3]). We randomly choose several well known SAT benchmark instances from formal verification, logic planning and random 3-SAT instances. The sizes of the subset to be eliminated are hand picked such that they do not cause blowup and take reasonable amount of time to finish. Table 1 shows the statistics of the original CNF formulas and the number of variables eliminated for each of the formulas. In Table 1 we also show the statistics of the final CNF formulas generated if we do not apply any CNF simplification techniques during the variable elimination process.

Table 2 shows the statistics for the variable elimination when we apply subsumption detection and removal technique described in section 3. Whenever a clause is added to the clause database, we check for subsumption and remove the subsumed clauses so that the CNF is subsumption free. The columns under “Total Added” are the number of clause and literals added to the clause database (i.e. number of clauses and literals passed to the routine *AddClauseRemoveSubsumption*). Columns under “Forward” and “Backward” show the number of clauses and literals removed due to forward and backward subsumption, respectively. “Final Result” shows the statistics of the final formula obtained. From the result we find that for this application, usually more clauses are forward subsumed than backward subsumed.

Table 3 shows the statistics for applying the DRF algorithm described in Section 3 in the variable elimination process. Again, under “Total Added” are the clauses and literals added to the CNF database, and the “Final Result” columns show the final CNF obtained. Compare Table 3 with table 2, we find that by maintaining the CNF to be DRF, we usually obtain smaller CNF formulas. Maintaining a CNF to be DRF is more expensive than subsumption removal. However, compare the run time under “Time” column for all three tables, we find that the performances are acceptable for this application because the great reduction in number of clauses offsets the costs for simplification.

5 Conclusions and Future Work

In this paper we describe an algorithm for subsumption detection and elimination on-the-fly, and an algorithm that makes a CNF database *decremental resolution free*. The purpose of this work is to automatically compact and simplify CNF formulas derived from operations such as resolution and solution enumeration. Such work is important because new applications of SAT solvers such as quantification elimination and abstraction refinement are performed by either resolution or solution enumeration.

Many things are unexplored in this paper. Our algorithms are not powerful enough to produce prime and irredundant CNF formulas. Traditional methods to generate prime and irredundant implicants are too expensive to be practical in this setting. How to balance the runtime cost and the quality of the result for on-the-fly CNF simplification is a very interesting problem worth much further investigation.

References

[1] João P. Marques-Silva and Karem A. Sakallah, “GRASP: A Search Algorithm for Propositional Satisfiability, In *IEEE Tran. on Computers*, *vol. 48, 506-521, 1999*

[2] L. Zhang and S. Malik, Towards Symmetric Treatment of Conflicts And Satisfaction in Quantified Boolean Satisfiability Solver, In *Proc. of 8th International Conference on Principles and Practice of Constraint Programming (CP2002)*. Ithaca, NY, Sept. 2002.

[3] M. Davis and H. Putnam, A computing procedure for quantification theory, *Journal of the ACM*, vol. 7, pp. 201-215, 1960.

[4] A. Biere. Resolve and Expand. In *Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, BC, Canada, 2004.

[5] Sathiamoorthy Subbarayan and Dhiraj K Pradhan, NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT instances, In *Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, BC, Canada, 2004.

[6] P. Chauhan, E. M. Clarke, D. Kroening, Using SAT Based Image Computation for Reachability Analysis, *Technical Report CMU-CS-03-151, Carnegie Mellon University*, July, 2003

[7] Johan de Kleer: An Improved Incremental Algorithm for Generating Prime Implicates. in *Proc. of the 10th National Conference on Artificial Intelligence (AAAI 1992)*, 1992

[8] Philippe Chatalic, Laurent Simon, Multi-Resolution on Compressed Sets of Clauses, in *Twelfth International Conference on Tools with Artificial Intelligence (ICTAI'00)*, 2000

[9] S. Minato, “Zero-Suppressed BDDs for Set Manipula-tion in Combinatorial Problems,” in *Proc. of the Design Automation Conference (DAC93), pp. 272-277*, 1993

[10] G. Hachtel and F. Somenzi, “Logic Sysntheiss and Verification Algorithms,” Kluwer Academic Publishers, 1996

[11] Hyeong Ju Kang, In-Cheol Park, SAT-Based Unbounded Symbolic Model Checking, in *Proc. 40th Design Automation Conference (DAC03)*, 2003

[12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT Solver, In *Proceedings of the Design Automation Conference, 2001*

Author Index