Constraints (2009) 14:325–356
DOI 10.1007/s10601-008-9062-z

Limitations of restricted branching in clause learning

Matti Järvisalo · Tommi Junttila

Published online: 7 January 2009

Abstract The techniques for making decisions, that is, branching, play a central role in complete methods for solving structured instances of constraint satisfaction problems (CSPs). In this work we consider branching heuristics in the context of propositional satisfiability (SAT), where CSPs are expressed as propositional formulas. In practice, there are cases when SAT solvers based on the Davis-Putnam-Logemann-Loveland procedure (DPLL) benefit from limiting the set of variables the solver is allowed to branch on to so called input variables which provide a strong unit propagation backdoor set to any SAT instance. Theoretically, however, restricting branching to input variables implies a super-polynomial increase in the length of the optimal proofs for DPLL (without clause learning), and thus input-restricted DPLL cannot polynomially simulate DPLL. In this paper we settle the case of DPLL with clause learning. Surprisingly, even with unlimited restarts, input-restricted clause learning DPLL cannot simulate DPLL (even without clause learning). The opposite also holds, and hence DPLL and input-restricted clause learning DPLL are polynomially incomparable. Additionally, we analyze the effect of input-restricted branching on clause learning solvers in practice with various structured real-world benchmarks.

This is an extended version of a paper [27] presented at the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007) in Providence, RI, USA. The first author gratefully acknowledges financial support from Helsinki Graduate School in Computer Science and Engineering, Academy of Finland (grants #211025 and #122399), Emil Aaltonen Foundation, Jenny and Antti Wihuri Foundation, Finnish Foundation for Technology Promotion TES, and Nokia Foundation. The second author gratefully acknowledges the financial support from Academy of Finland (grant #112016).

M. Järvisalo (✉) · T. Junttila
Department of Information and Computer Science, Helsinki University of Technology TKK, PO Box 5400, 02015 TKK, Finland
e-mail: matti.jarvisalo@tkk.fi

T. Junttila
e-mail: tommi.junttila@tkk.fi

Keywords Propositional satisfiability · Branching heuristics · Clause learning · DPLL · Proof complexity · Problem structure · Backdoor sets

1 Introduction

Modern complete satisfiability (SAT) solvers (such as [18, 22, 39, 40] among others) provide an efficient way of solving various real-world problems as propositional satisfiability. Typical SAT solvers aimed at solving such structured problems are based on the conjunctive normal form (CNF) level *Davis-Putnam-Logemann-Loveland* procedure (DPLL) [16, 17] and incorporate techniques such as *intelligent branching heuristics*, *randomization* and *restarts* [23], and *clause learning* [39] for boosting search efficiency.

Branching heuristics, that is, deciding on which variable to next set a value during search, play an important role in the efficiency of complete SAT methods aimed at solving typically very large real-world problem instances. Intuitively, the inherent structure of the problem domain is reflected in individual variables in the SAT encoding, and making decisions on structurally irrelevant variables may have an exponential effect on the running times of SAT solvers.

In SAT-based approaches to structured problems such as bounded model checking [10] and automated planning [33], the CNF encoding is often derived from a transition relation, where the behavior of the underlying system is dependent on the *input*—initial state, nondeterministic choices due to external control, et cetera—of the system. Empirical case studies [14, 19, 20, 45] have shown that, in some cases, SAT solvers benefit from restricting the variables the solver is allowed to branch on to so called *input* (or *independent*) *variables*, corresponding to the input of the underlying system. By noticing that the system behavior is determined by its input, it is in fact the case that all variables in the SAT encoding of the system can be assigned through unit propagation once all input variables have been assigned values. In other words, the set of input variables is a *strong unit propagation backdoor set* [51]—although possibly not of *minimum* cardinality. Hence DPLL remains complete even if branching is restricted to the set of input variables alone. Intuitively, this drops the raw search space size from 2^N to 2^I with $I \ll N$, where I and N are the number of input variables and all variables in the CNF encoding, respectively.

From another point of view to the effects of different techniques for branching, one can investigate the *best-case* performance of SAT algorithms through *proof complexity* [13], by studying the relative power of their underlying inference systems (or *proof systems*) in terms of the shortest existing proofs in the systems. For two proof systems, S and S', we say that S' *(polynomially) simulates* S if, for all infinite families $\{F_n\}$ of unsatisfiable CNF formulas, there is a polynomial that bounds for all F_n the length of the shortest proofs in S' w.r.t. the length of the shortest proofs in S. If S' simulates S and vice versa, then S and S' are *polynomially equivalent*. If S' cannot simulate S and vice versa, then S and S' are *incomparable*. From the practical point of view, if S' cannot simulate S, we know that any implementation of S' can suffer a notable decrease in efficiency compared to implementations of S. For example, through a formal characterization of DPLL with clause learning, called CL, Beame et al. [9] show that CL can provide superpolynomially shorter proofs than DPLL, and thus DPLL cannot simulate CL.

Considering restricting branching in DPLL algorithms to input variables, a natural question to ask is *whether the power of the underlying inference systems of* DPLL-*based solvers is affected by the input-restriction*. For DPLL without clause learning, this question is answered in [29]: input-restricted DPLL cannot simulate DPLL.

In this paper we settle the case of input-restricted CL: it turns out that input-restricted CL cannot simulate CL. This implies that all implementations of clause learning DPLL, even with optimal heuristics, have the potential of suffering a notable efficiency decrease if branching is restricted to input variables. In fact, we show that even with unlimited restarts and the ability to create conflicts at will, input-restricted CL cannot even simulate the basic DPLL *without clause learning*. This is surprising, since the unrestricted version of this variant of CL can efficiently simulate (general) Resolution [9], being thus very powerful compared to DPLL. Additionally, we evaluate the effect of input-restricted branching on clause learning with various structured real-world benchmarks, with possible explanations for the reasons why input-restricted branching can in fact hinder the efficiency of typical clause learning solvers.

As preliminaries, in Section 2 we define Boolean circuits, which we use for representing general propositional formulas, and discuss their relation to CNF formulas. We then review the Resolution proof system and characterizations of DPLL and CL, and discuss known results concerning their relative efficiency (Section 3). Section 4 concentrates on the tight correspondence between a constrained Boolean circuit and its CNF translation from the viewpoint of DPLL and clause learning, which is of value in presenting the theoretical results of this work. The main theoretical and experimental contributions of this paper are presented in Sections 5 and 6, respectively.

2 Propositional satisfiability and constrained Boolean circuits

In this section we review basic concepts related to propositional satisfiability and define constrained Boolean circuits which we use as the representation form for structured formulas. We also discuss the relationship between constrained Boolean circuits and clausal propositional (CNF) formulas, and present the translation from constrained Boolean circuits to CNF which is applied in this work.

2.1 Propositional satisfiability

Given a Boolean variable x, there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\neg x$, where $\neg$ is the logical negation (not). As usual, we identify $\neg\neg x$ with x. A *clause* is a disjunction ($\vee$, or) of distinct literals and a CNF formula is a conjunction ($\wedge$, and) of clauses. When convenient, we view a clause as a finite set of literals and a CNF formula as a finite set of clauses; e.g. the formula $(a \vee \neg b) \wedge (\neg c)$ can be written as $\{\{a, \neg b\}, \{\neg c\}\}$. The sets of variables appearing as positive and negative literals in a CNF formula F are denoted by $\mathsf{vars}^+(F)$ and $\mathsf{vars}^-(F)$, respectively, and the set of variables by $\mathsf{vars}(F)$; for a clause C, $\mathsf{vars}^+(C)$, $\mathsf{vars}^-(C)$, and $\mathsf{vars}(C)$ are defined similarly.

Given a CNF formula F, a (partial) *assignment* for F is a (partial) function $\tau : \mathsf{vars}(F) \to \{\mathbf{t}, \mathbf{f}\}$, where $\mathbf{t}$ and $\mathbf{f}$ stand for *true* and *false*, respectively. With a slight abuse of notation, if $\tau(x) = v$, then $\tau(\neg x) = \neg v$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$. A clause is *satisfied* by τ if it contains at least one literal l such that $\tau(l) = \mathbf{t}$. If $\tau(l) = \mathbf{f}$ for every literal l in a clause, the clause is *falsified* by τ. An assignment τ *satisfies* a CNF formula it satisfies every clause in the formula. A formula is *satisfiable* if there is an assignment that satisfies it, and *unsatisfiable* otherwise.

2.2 Constrained Boolean circuits

The correspondence between system input of a real-world problem and propositional variables in a CNF encoding is not evident. However, in SAT-based approaches, direct CNF encodings of a problem domain are rarely used: the problem at hand is typically encoded with a general propositional formula ϕ, which is then translated into an equi-satisfiable CNF formula by introducing additional variables for the sub-formulas of ϕ. *Boolean circuits* (see e.g. [42]) offer a natural way of presenting propositional formulas in a compact DAG-like structure with *sub-formula sharing*, which helps in lowering the number of additional variables needed. Additionally, the system input of the original problem is presented by *input gates* in Boolean circuits.

A Boolean circuit over a finite set G of *gates* is a set $\mathscr{C}$ of equations of form $g := f(g_1, \ldots, g_n)$, where $g, g_1, \ldots, g_n \in G$ and $f : \{\mathbf{f}, \mathbf{t}\}^n \to \{\mathbf{f}, \mathbf{t}\}$ is a Boolean function, with the additional requirements that (i) each $g \in G$ appears at most once as the left hand side in the equations in $\mathscr{C}$, and (ii) the underlying directed graph

$$\left\langle G, E(\mathscr{C}) = \left\{\langle g', g\rangle \in G \times G \mid g := f(\ldots, g', \ldots) \in \mathscr{C}\right\}\right\rangle$$

is acyclic. If $\langle g', g\rangle \in E(\mathscr{C})$, then g' is a *child* of g and g is a *parent* of g'. Similarly, if there is a non-empty path from a gate g' to a gate g in $\langle G, E(\mathscr{C})\rangle$, then g' is a *descendant* of g. If $g := f(g_1, \ldots, g_n)$ is in $\mathscr{C}$, then g is an f-gate (or of type f), otherwise it is an *input gate*. A gate with no parents is an *output gate*. A (partial) assignment for $\mathscr{C}$ is a (partial) function $\tau : G \to \{\mathbf{f}, \mathbf{t}\}$. An assignment τ is *consistent* with $\mathscr{C}$ if $\tau(g) = f(\tau(g_1), \ldots, \tau(g_n))$ for each $g := f(g_1, \ldots, g_n)$ in $\mathscr{C}$. Note that a circuit with I input gates has 2^I consistent assignments.

A *constrained Boolean circuit* $\mathscr{C}^\tau$ is a pair $\langle \mathscr{C}, \tau\rangle$, where $\mathscr{C}$ is a Boolean circuit and τ is a partial assignment for $\mathscr{C}$. With respect to a $\langle \mathscr{C}, \tau\rangle$, each $\langle g, v\rangle \in \tau$ is a *constraint*, and g is *constrained* to v if $\langle g, v\rangle \in \tau$. An assignment τ' *satisfies* $\mathscr{C}^\tau$ if (i) it is consistent with $\mathscr{C}$, and (ii) it respects the constraints in τ, meaning that for each gate $g \in G$, if $\tau(g)$ is defined, then $\tau'(g) = \tau(g)$. If some assignment satisfies $\mathscr{C}^\tau$, then $\mathscr{C}^\tau$ is *satisfiable* and otherwise *unsatisfiable*.

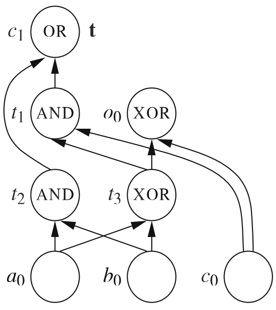
In the following, we will apply the following Boolean functions as gate types. Notice that this set of is sufficient for representing all Boolean functions, and on the other hand, enough for describing the constructions applied in this paper in an intuitive way.

– NOT(g) evaluates to $\mathbf{t}$ if and only if g evaluates to $\mathbf{f}$.
– OR$(g_1, \ldots, g_n)$ evaluates to $\mathbf{t}$ if and only if at least one of $g_1, \ldots, g_n$ evaluates to $\mathbf{t}$.
– AND$(g_1, \ldots, g_n)$ evaluates to $\mathbf{t}$ if and only if all $g_1, \ldots, g_n$ evaluate to $\mathbf{t}$.
– XOR(g_1, g_2) evaluates to $\mathbf{t}$ if and only if exactly one of g_1, g_2 evaluates to $\mathbf{t}$.

Example 1 A Boolean circuit $\mathscr{C}^\tau$ and its graphical representation are shown in Fig. 1. The circuit models a full-adder with the constraint that the carry-out bit

Fig. 1 A constrained Boolean circuit $\mathscr{C}^{\tau}$ and its graphical representation

$$\begin{aligned}\mathscr{C} = \{c_1 &:= \text{OR}(t_1, t_2)\\ t_1 &:= \text{AND}(t_3, c_0)\\ o_0 &:= \text{XOR}(t_3, c_0)\\ t_2 &:= \text{AND}(a_0, b_0)\\ t_3 &:= \text{XOR}(a_0, b_0)\}\\ \tau = \{\langle c_1, \mathbf{t}\rangle\}&\end{aligned}$$



c_1 is **t**. A satisfying truth assignment for the circuit is $\tau' = \{\langle c_1, \mathbf{t}\rangle, \langle t_1, \mathbf{t}\rangle, \langle o_0, \mathbf{f}\rangle, \langle t_2, \mathbf{f}\rangle, \langle t_3, \mathbf{t}\rangle, \langle a_0, \mathbf{t}\rangle, \langle b_0, \mathbf{f}\rangle, \langle c_0, \mathbf{t}\rangle\}$.

For notational convenience, when well-defined, the *join* of two constrained circuits, $\mathscr{A}^{\tau} = \langle \mathscr{A}, \tau\rangle$ and $\mathscr{B}^{\theta} = \langle \mathscr{B}, \theta\rangle$, is $\mathscr{A}^{\tau} \cup \mathscr{B}^{\theta} = \langle \mathscr{A} \cup \mathscr{B}, \tau \cup \theta\rangle$. When applying the join, we will always make sure that the result is a well-defined constrained Boolean circuit. This means that the requirements (i) on unique definition and (ii) on acyclicity above are met, and that $\tau \cup \theta$ is a (possibly partial) function.

2.3 Translating Boolean circuits to CNF

In order to exploit clausal SAT solvers in solving instances of Boolean circuit satisfiability, the circuit in question has to be translated to CNF. In this work we apply the standard "Tseitin-style" [47] translation. First, a variable $\tilde{g}$ is introduced for each gate g. For encoding the functionalities of gates, the idea is to represent the logical equivalence $g \Leftrightarrow f(g_1, \ldots, g_n)$ as clauses; hence for each $g := f(g_1, \ldots, g_n)$ the corresponding introduced clauses are as shown in Table 1. Similarly, a unit clause is added for each constraint $\langle g, v\rangle \in \tau$ as shown in Table 1. Given a constrained Boolean circuit $\mathscr{C}^{\tau}$, we will denote its CNF translation by $\mathsf{cnf}(\mathscr{C}^{\tau})$.

2.4 CNF formulas as constrained circuits

Any CNF formula $F = \{C_1, \ldots, C_k\}$ can naturally be seen as a Boolean circuit. Basically, F is a Boolean circuit with an AND of ORs which represent the clauses. Formally, $\mathsf{circuit}(F) := \langle \mathscr{C}, \tau\rangle$ is defined by associating an input gate x with each

Table 1 CNF translation for constrained Boolean circuits

Gate or constraint	Clauses
$g := \text{XOR}(g_1, g_2)$	$(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \neg\tilde{g}_2)$, $(\neg\tilde{g} \vee \tilde{g}_1 \vee \tilde{g}_2)$, $(\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2)$, $(\tilde{g} \vee \tilde{g}_1 \vee \neg\tilde{g}_2)$
$g := \text{OR}(g_1, \ldots, g_n)$	$(\neg\tilde{g} \vee \tilde{g}_1 \vee \cdots \vee \tilde{g}_n)$, $(\tilde{g} \vee \neg\tilde{g}_1)$,…,$(\tilde{g} \vee \neg\tilde{g}_n)$
$g := \text{AND}(g_1, \ldots, g_n)$	$(\neg\tilde{g} \vee \tilde{g}_1)$,…,$(\neg\tilde{g} \vee \tilde{g}_n)$, $(\tilde{g} \vee \neg\tilde{g}_1 \vee \cdots \vee \neg\tilde{g}_n)$
$g := \text{NOT}(g_1)$	$(\neg\tilde{g} \vee \neg\tilde{g}_1)$, $(\tilde{g} \vee \tilde{g}_1)$
$\langle g, \mathbf{t}\rangle \in \tau$	$(\tilde{g})$
$\langle g, \mathbf{f}\rangle \in \tau$	$(\neg\tilde{g})$

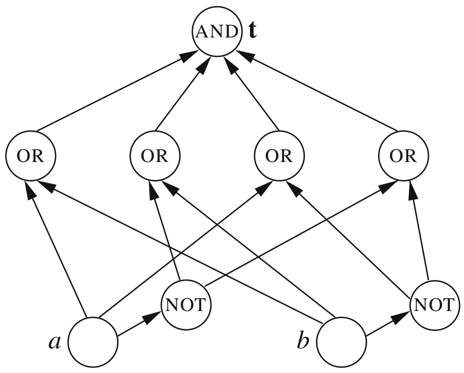


Fig. 2 The constrained Boolean circuit $\mathsf{circuit}(\{\{a,b\},\{a,\neg b\},\{\neg a,b\},\{\neg a,\neg b\}\})$

variable $x \in \mathsf{vars}(F)$, a NOT-gate $g_{\neg x}$ with each $x \in \mathsf{vars}^{-}(F)$, an OR-gate g_{C_i} with each clause $C_i \in F$, an AND-gate g_F with F, and by setting $\tau = \{\langle g_F, \mathbf{t}\rangle\}$ and

$$\begin{aligned}\mathscr{C} := & \left\{g_F := \textsc{and}(g_{C_1},\ldots,g_{C_k})\right\} \cup \left\{g_{\neg x} := \textsc{not}(x) \mid x \in \mathsf{vars}^{-}(F)\right\} \\ \cup & \left\{g_{C_i} := \textsc{or}(\alpha(l_{i,1}),\ldots,\alpha(l_{i,n_i})) \mid C_i = \left\{l_{i,1},\ldots,l_{i,n_i}\right\} \in F\right\}\end{aligned}$$

where $\alpha(\neg x) = g_{\neg x}$ and $\alpha(x) = x$ for each $x \in \mathsf{vars}(F)$.

Example 2 The constrained Boolean circuit $\mathsf{circuit}(F)$ for the unsatisfiable CNF formula $F = \{\{a,b\},\{a,\neg b\},\{\neg a,b\},\{\neg a,\neg b\}\}$ is shown in Fig. 2.

3 Proof systems for CNF formulas

In this section we discuss the propositional proof systems of interest in the context of this work, with known results on their relative efficiency. First, we formally define propositional proof systems and the necessary proof complexity theoretic notions. We then review the well-known Resolution proof system and some of its refinements. After this, we concentrate on the Davis–Putnam–Logemann–Loveland (or DPLL) procedure [16, 17] and the additional techniques applied in typical DPLL-based SAT solvers today—most importantly, clause learning. In doing so, we go through characterizations of DPLL (with and without clause learning) as proof systems, which we will apply in the theoretical part of the work.

3.1 Propositional proof systems and complexity

Formally, a *propositional proof system* [13] is a polynomial-time computable predicate S such that a propositional formula F is unsatisfiable if and only if there is a *proof* p for which $S(F, p)$ holds. Thus a proof p of F is a *certificate* of the unsatisfiability of F, and a proof system is a polynomial-time procedure for checking the validity of proofs in a certain format.

While proof checking is efficient, finding short proofs may be difficult, or, generally, impossible since short proofs may not exist for too weak a proof system. As a measure of hardness of proving unsatisfiability of a CNF formula F in a proof system S, the *(proof) complexity* $C_S(F)$ of F in S is the *length* of the shortest proof of F in S. For a family $\{F_n\}$ of unsatisfiable CNF formulas over an increasing number of variables, the (asymptotic) complexity of $\{F_n\}$ is measured with respect to the number of clauses in F_n.

For two proof systems, S and S', we say that S' *(polynomially) simulates* S if for all families $\{F_n\}$ there is a polynomial p such that $C_{S'}(F_n) \leq p(C_S(F_n))$ for all F_n. If S simulates S' and vice versa, then S and S' are *polynomially equivalent*. If there is a family $\{F_n\}$ for which S' does not polynomially simulate S, we say that $\{F_n\}$ *separates* S from S'. If S can be separated from S' and vice versa, then S and S' are *incomparable*. Notice that polynomial simulation gives a partial order for proof systems based on their relative power.

With these definitions, in order to show that a proof system S cannot simulate another system S', it suffices to exhibit an infinite family $\{F_n\}$ of unsatisfiable formulas over an increasing number of variables, such that the minimum length proofs in S for $\{F_n\}$ are asymptotically superpolynomially longer than the minimum length proofs in S' with respect to the number of clauses in F_n. It is worth noticing that, from this basic proof complexity theoretic point of view only *unsatisfiable* formulas (and hence proofs of unsatisfiability) are of interest. Although exponential lower bounds for DPLL on families of *satisfiable* formulas have been shown in restricted probabilistic contexts [1, 2, 4, 41], a satisfying truth assignment acts as a polynomial length witness for the satisfiability of an arbitrary satisfiable formula F.

3.2 Resolution

The well-known Resolution proof system [44] (RES) is based on the *resolution rule*. Let C, D be clauses, and x a Boolean variable. The resolution rule is

$$\frac{\{x\} \cup C \quad \{\neg x\} \cup D}{C \cup D}$$

or, in other words, we can *directly derive* $C \cup D$ from $\{x\} \cup C$ and $\{\neg x\} \cup D$ by *resolving on* x. For a given CNF formula F, a RES *derivation of a clause* C from F is a sequence of clauses $\pi = (C_1, C_2, \ldots, C_m = C)$, where each C_i, $1 \leq i \leq m$, is either (i) a clause in F (an *initial clause*), or (ii) directly derived with the resolution rule from two clauses C_j, C_k where $1 \leq j, k < i$ (a *derived clause*). The *length* of π is m, the number of clauses occurring in it. A RES *proof (for the unsatisfiability)* of a CNF formula F is any RES derivation of the empty clause $\emptyset$ from F.

Any RES derivation $\pi = (C_1, C_2, \ldots, C_m)$ can be presented as a directed acyclic graph, in which the leafs are initial clauses and the other nodes represent derived clauses. The edge relation is defined so that there are edges from C_i and C_j to C_k, if and only if C_k has been directly derived from C_i and C_j using the resolution rule. Many *refinements of* Resolution, in which the structure of RES proofs is restricted, have been proposed and studied. Here of particular interest is *Tree-like Resolution* (T-RES), with the requirement that proofs are representable as trees. This implies that a derived clause, if used multiple times in the proof, must be derived anew each time starting from initial clauses.

3.2.1 Lower bounds in RES *and its refinements*

Super-polynomial (and even exponential) lower bounds on proof lengths in RES have been shown for various families of CNF formulas, see [3, 6, 7, 11, 15, 24, 47, 48] for examples. Among the most studied such families is the *pigeon-hole principle*, which states that there is no injective mapping from an m-element set into an n-element set if $m > n$ (that is, m pigeons cannot sit in fewer than m holes so that every pigeon has its own hole). We will consider the case $m = n + 1$ encoded as the CNF formula

$$\mathrm{PHP}_n^{n+1} := \bigwedge_{i=1}^{n+1} \left(\bigvee_{j=1}^{n} p_{i,j} \right) \wedge \bigwedge_{j=1}^{n} \bigwedge_{i=1}^{n} \bigwedge_{i'=i+1}^{n+1} (\neg p_{i,j} \vee \neg p_{i',j}),$$

where each $p_{i,j}$ is a Boolean variable with the interpretation "$p_{i,j}$ is $\mathbf{t}$ if and only if the ith pigeon sits in the jth hole".

Theorem 1 ([24]) *There is no polynomial length* RES *proof of* PHP_n^{n+1}.

It is also known that T-RES is a proper refinement of RES in the sense that T-RES cannot polynomially simulate RES.

Theorem 2 ([21, 49]) T-RES *cannot polynomially simulate* RES.

This originates from the facts that *regular resolution* cannot simulate RES [5, 21], and T-RES in turn cannot simulate regular resolution [49].

3.3 The Davis–Putnam–Logemann–Loveland procedure

Most modern complete SAT solvers are based on the Davis–Putnam–Logemann–Loveland (or DPLL) procedure [16, 17]. Given a CNF formula F as input, DPLL is a depth-first search procedure building a partial assignment τ for the variables in F through (i) *branching* and (ii) *unit propagation* (UP). In branching, the current assignment τ is extended with the assignment (*decision*) $\langle x, v \rangle$, where v is either $\mathbf{f}$ or $\mathbf{t}$, for some unassigned variable x. Unit propagation refers to applying the *unit clause rule*. The unit clause rules states that if there is a clause $(l_1 \vee \cdots \vee l_k \vee l) \in F$ such that $\tau(l_i) = \mathbf{f}$ for each $1 \leq i \leq k$, the current partial assignment τ can be extended with $\langle l, \mathbf{t} \rangle$.

An assignment is extended until (i) some variable x would be assigned both $\mathbf{f}$ and $\mathbf{t}$ (a *conflict* is reached, with x as the *conflict variable*) or (ii) τ satisfies F (in which case DPLL terminates). In case (i), non-clause learning DPLL solvers *backtrack* to the last branching decision which has not been backtracked upon, undoing all assignments made by UP after the particular decision, and flip the decision. DPLL terminates on an unsatisfiable CNF formula when there are no untried branches left.

From the proof theoretic point of view, DPLL can be seen as a tableau proof system with two rules: the *branching rule* and the unit clause rule. The branching rule, corresponding to branching on a variable x, extends the branch into two branches, one of which is extended with the entry x and the other with $\neg x$. The unit clause rule, defined above, is similarly applied by extending the branch with l. As typical, a branch is (fully) extended until we have both of the entries x and $\neg x$ for some variable, or no new entries can be generated with the branching and unit clause rules. From an algorithmic point of view, the choice of in which order branches are extended is part of the solver strategy, and based on a *decision heuristic*. The other branch resulting from the particular application of the branching rule is handled through backtracking. With this intuition, it is clear that a search tree traversed by a DPLL algorithm corresponds to a binary tableau proof, having the form of a binary tree, with all branches fully extended. Hence, a DPLL *proof* will here be such a tableau proof. The length of a DPLL proof is defined as the number of applications of the branching rule in the proof.

One-step lookahead (see, e.g., [37]) is an often implemented technique in (non-clause learning) DPLL algorithms. In one-step lookahead, if there is an assignment v to a currently unassigned variable x such that the current assignment τ with the addition of $\langle x, v\rangle$ leads to a conflict using unit propagation, then x is immediately assigned the value $\neg v$. This technique does not add to the strength of DPLL, since the same effect can obviously be accomplished by branching on x.

It is well-known that DPLL and T-RES can polynomially simulate each other (see [8] for example). One can show that for any unsatisfiable CNF formula, a minimum length DPLL proof, with applications of the unit clause rule "simulated by branching", always corresponds one-to-one with a minimum length T-RES proof, and vice versa.

Fact 1 DPLL *and* T-RES *are polynomially equivalent.*

3.3.1 Implication graphs

Implication graphs capture the ways of deriving values for variables with the unit clause rule from assignments made by branching. We will apply this concept in the following for defining clause learning. However, first we need some additional terminology.

A *stage* of DPLL on a CNF formula F is characterized by the *decision literals* in the branch. Considering an arbitrary branch, the variables assigned by branching are called *decision variables* and those assigned values by UP are *implied variables*, with analogous definitions for *decision literals* and *implied literals*. The *decision level of a decision variable x* is one more than the number of decision variables in the branch before branching on x. The *decision level of an implied variable x* is the number of decision variables in the branch when x is assigned a value. The decision level of DPLL at any stage is the number of decision variables in the branch.

For a given CNF formula F and a set of literals L, we denote by $F, L \vdash_{\mathrm{UP}} l$ the fact that l can be deduced from F and L by iteratively applying the unit clause rule.

Definition 1 For a CNF formula F, the *implication graph* $G = \langle V, E\rangle$ at a given stage of DPLL with the set of decision literals D is a directed graph. The set of nodes is defined as

$$V = \{\Lambda\} \cup D \cup \{l \mid F, D \vdash_{\mathrm{UP}} l\},$$

where Λ is a special *conflict node*, and the edge relation is

$$E = \{\langle \neg l_i, l\rangle \mid \{l_1, \ldots, l_k, l\} \in F \text{ and } \neg l_1, \ldots, \neg l_k \in V\}$$
$$\cup \{\langle x, \Lambda\rangle, \langle \neg x, \Lambda\rangle \mid x, \neg x \in V\}.$$

For a given implication graph, a variable x with both $x, \neg x \in V$ is called a *conflict variable*, and $x, \neg x$ are *conflict literals*. An implication graph contains a conflict if it contains a conflict variable; DPLL has a conflict at a given stage if the implication graph at the stage contains a conflict.

3.4 DPLL with clause learning and modern SAT solvers

Clause learning DPLL algorithms differ from non-clause learning algorithms in what happens when reaching a conflict. If a conflict is reached without any branching, DPLL (with or without clause learning) determines the formula F unsatisfiable. In other cases, non-clause learning DPLL algorithm perform simple backtracking as previously explained. In clause learning DPLL algorithms, however, the conflict is *analyzed*, and a *learned clause* (or *conflict clause*), which describes the "cause" of the conflict, is added to F. After this the search is continued typically by applying *non-chronological backtracking* (or *conflict-driven backjumping*) for backtracking to an earlier decision level that "caused" the conflict. Conflict-driven backjumping results in the fact that, as opposed to the basic backtracking in DPLL, the other branch (opposite value) of decision variables is not necessarily forced systematically when backtracking. In other words, branching in clause learning DPLL is seen simply as assigning values to unassigned variables, rather than as a branching rule in which by branching on a variable x the current branch is always extended into two branches, one with x and the other with $\neg x$.

3.4.1 Conflict graphs and conflict analysis

Similarly as with DPLL, the *stage* of a clause learning DPLL algorithm is characterized by the set of decision literals. At a given stage of a clause learning DPLL algorithm, a clause is called *known* if it either appears in the original CNF formula F or has been learned earlier during the search. Conflict analysis is based on a *conflict graph*, which captures one way of reaching the conflict at hand from the decision variables by using the unit clause rule on known clauses.

Definition 2 Given an implication graph G, a *conflict graph* $H = (V, E)$ based on G is any acyclic subgraph of G having the following properties.

1. H contains Λ and exactly one conflict literal pair $x, \neg x$.
2. All nodes in H have a path to Λ.
3. Every node $l \in V \setminus \{\Lambda\}$ either corresponds to a decision literal or has precisely the nodes $\neg l_1, \neg l_2, \ldots, \neg l_k$ as predecessors where $\{l_1, l_2, \ldots, l_k, l\}$ is a known clause.

A conflict graph describes a single conflict and contains only decision and implied literals that can be used in reaching the conflict when applying the unit clause rule *in some order*. Hence the way of implementing unit propagation in a solver has an effect

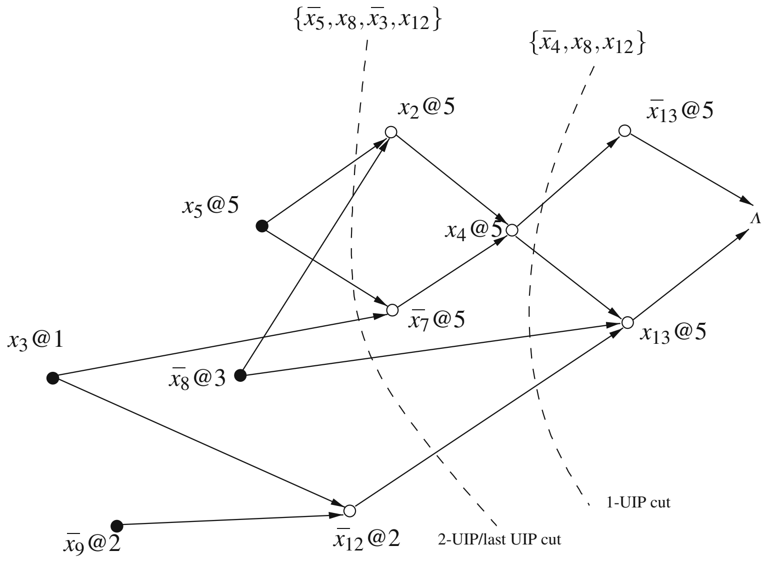


Fig. 3 Example of a conflict graph, and two possible conflict cuts

on the choice of the conflict graph. The acyclicity of conflict graphs results from the fact that unit propagation is not used to rederive already assigned literals.

Conflict clauses are associated with *cuts* in a conflict graph. Fix a conflict graph contained in an implication graph with a conflict. A *conflict cut* is any cut in the conflict graph with all the decision variables on one side (the *reason side*) and, in addition to Λ, at least one conflict literal on the other side (the *conflict side*). Those nodes on the reason side with at least one edge going to the conflict side in a conflict cut form a cause of the conflict. With the associated literals set to **t**, UP can arrive at the conflict at hand. The disjunction of the negations of these literals form the *conflict clause associated with the conflict cut*. The strategy for fixing a conflict cut is called the *learning scheme*. A learning scheme which always learns a currently unknown clause is called *non-redundant*.

Example 3 A hypothetical conflict graph is illustrated in Fig. 3. Decision literals are represented with filled circles, and implied literals with hollow circles. The decision level d of each literal l is presented with the label $l@d$. For example, the conflict variable x_{13} is at decision level 5. Notice that since the literals at decision level 4 are missing from this conflict graph, they are not part of the reason for the particular conflict. In the figure two possible conflict cuts are shown with the associated conflict clauses.

3.4.2 Unique implication points, conflict-driven backjumping, and CL *proofs*

Typically implemented clause learning schemes are based on *unique implication points* (UIPs) [39]. A UIP in a conflict graph is a node u on the maximum decision level d such that all paths from the decision variable x at level d to Λ go through u.

Such a u always exists as x satisfies this condition. Intuitively, u is a *single* reason for the conflict at level d. Thus one can always choose a conflict cut that results in a conflict clause with a UIP as the only variable from the maximum decision level. Such a conflict clause has the property that the UIP variable can be immediately set to the value opposite to the current assignment using the unit clause rule when backtracking (the phrase "the UIP is *asserted*" is sometimes used). Furthermore, UIP learning schemes enable *conflict-driven backtracking* (or *backjumping*), in which DPLL backtracks to the maximum decision level of the variables other than the UIP in a conflict clause. A popular version of UIP learning is the 1-UIP scheme, where a conflict cut is chosen so that the UIP closest to Λ will be in the associated conflict clause. Different learning schemes are evaluated in [52], showing the robustness of the 1-UIP scheme in practice.

Example 4 Recall the conflict graph in Fig. 3. The 1-UIP in this graph is the literal x_4. One conflict cut corresponding to the 1-UIP learning scheme is the cut labeled "1-UIP cut". The cut labeled "2-UIP cut/last UIP cut" can result from applying the *second UIP scheme* in which a conflict clause with the UIP second closest to Λ is chosen. In this example, the "2-UIP cut" is at the same time a cut that can result from applying the *last UIP scheme* in which a cut with the decision literal on the maximum decision level as the UIP is chosen.

For investigating the efficiency of clause learning DPLL in proof complexity theoretic terms, we need to have a proof system characterization of clause learning DPLL algorithms. We will use the following characterization, referred to as the CL *proof system*. Here we loosely follow the characterization of [9]. A clause learning proof (or CL proof) induced by a learning scheme S is constructed by applying branching and the unit clause rule, using S to learn conflict clauses when conflicts are reached, so that in the end, a *conflict can be reached at decision level zero*. When a conflict cut with a UIP is selected, it is possible to apply conflict-driven backjumping based on the conflict clause. Otherwise, simple backtracking is applied. Notice that this definition allows even the most general *nondeterministic learning scheme* [9], in which the conflict cut is selected nondeterministically from the set of all possible conflict cuts related to the conflict graph at hand.

Hence, a CL proof can be seen as a tree in which the traversal order is marked in the nodes. Each leaf node in the tree is labeled with a conflict graph, a conflict cut in the graph, and the decision level onto which to backjump. Now, the proof system CL consists of CL proofs under any learning scheme. The length of a CL proof is the number of branching decisions.

While the practical efficiency gains of implementing clause learning into DPLL-based algorithms are well-established, the first formal study on the power of clause learning is [9]: CL can provide exponentially shorter proofs than T-RES even if no restarts are allowed. Thus we have the following corollary.

Corollary 1 (of Theorem 1 in [9]) DPLL *cannot polynomially simulate* CL.

3.4.3 Restarts and the CL-- *proof system*

Restarting is an additional technique often implemented in modern solvers. When a restart occurs, the decisions and unit propagations made so far are undone, and the search continues from decision level zero. The clauses learned so far remain known after the restart. Intuitively, restarts help in escaping from getting stuck in hard-to-prove sub-formulas. In practice, the choice of when and how often to restart is part of the strategy of a solver. When any number of restarts are allowed during search, we say that CL has *unlimited restarts*. For a recent investigation into the effect of restarts on the efficiency of clause learning DPLL algorithms, see [25].

Beame et al. [9] define CL-- as CL with branching allowed also on already assigned values. Although being non-typical in practice, this enables creating immediate conflicts at will. Although it is not known whether CL can simulate RES, it has been shown that this is true for CL-- using unlimited restarts.

Theorem 3 ([9]) RES *and* CL-- *with unlimited restarts and any non-redundant learning scheme are polynomially equivalent.*

We note that the proof of this theorem in [9] relies on the fact that unit propagation is seen as applications of the unit clause rule, and hence the rule can also be left unapplied when convenient. This is non-typical for implementations of clause learning DPLL; they usually apply unit propagation eagerly whenever possible.

4 Relating CNF proof systems and circuit structure

A key element in this work is the tight correspondence between a constrained Boolean circuit $\mathscr{C}^\tau$ and its CNF translation $\mathsf{cnf}(\mathscr{C}^\tau)$. In this section we review details on the correspondence of deduction in the CNF translation of a Boolean circuit with the original circuit structure, and on how branching in DPLL and CL can be restricted based on the original circuit structure. These details play an integral role in the theoretical results presented in the next section.

4.1 Unit propagation on the level of circuits

As there is a one-to-one relationship between the gates in a constrained Boolean circuit $\mathscr{C}^\tau$ and the variables in the corresponding CNF formula $\mathsf{cnf}(\mathscr{C}^\tau)$, the variables can be thought to inherit the structural properties of the gates. For example, an *input variable* is a variable that corresponds to an input gate in the original Boolean circuit, and we will take the liberty of using the terms “gate” and “variable” synonymously. Furthermore, since the CNF translation in Table 1 encodes in a natural way the semantics of the gates, unit propagation in the CNF formula can be seen as working on the level of the circuit. A further discussion on this can be found e.g. in [29], using a unit propagation equivalent characterization of Boolean constraint propagation as deduction rules for circuits [31]. Basically, such circuit level Boolean constraint propagation can set a value on a gate if and only if unit clause propagation can set a value on the corresponding Boolean variable in the CNF translation. For example, consider the gate $g := \textsc{and}(g_1, g_2)$ and its CNF translation $(\neg\tilde{g} \vee \tilde{g}_1) \wedge (\neg\tilde{g} \vee \tilde{g}_2) \wedge (\tilde{g} \vee \neg\tilde{g}_1 \vee \neg\tilde{g}_2)$. Now whenever the gate g_2 is assigned to $\mathbf{f}$, the gate g can be propagated to $\mathbf{f}$ by the semantics of $\textsc{and}$. On the CNF level, we can equivalently propagate the variable $\tilde{g}$ to $\mathbf{f}$ by applying the unit clause rule whenever the variable $\tilde{g}_2$ is assigned to $\mathbf{f}$ through the clause $(\neg\tilde{g} \vee \tilde{g}_2)$. The same kind of equivalent behaviour is noticed in a "top-down" fashion when assigning the gate g to $\mathbf{t}$: on the circuit-level, the gates g_1 and g_2 can be propagated to $\mathbf{t}$, and on the CNF level we can equivalently propagate the variables $\tilde{g}_1$ and $\tilde{g}_2$ to $\mathbf{t}$ through the clauses $(\neg\tilde{g} \vee \tilde{g}_1)$ and $(\neg\tilde{g} \vee \tilde{g}_2)$, respectively, by applying the unit clause rule whenever the variable $\tilde{g}$ is assigned to $\mathbf{t}$.

Hence we will also take the liberty of saying that unit propagation sets a value on a gate when referring to unit propagation setting a value on the corresponding Boolean variable in the CNF translation. Similarly, we *branch on a gate* when referring to branching on the corresponding Boolean variable. Correspondingly, a DPLL or CL proof of a constrained circuit $\mathcal{C}^\tau$ means a proof of the translation $\mathsf{cnf}(\mathcal{C}^\tau)$.

Since unit propagation can be also seen as Boolean constraint propagation on the level of constrained circuits, DPLL can also be implemented as a circuit level procedure, see, e.g., [31, 35, 38, 46]. Since conflict graphs are based on how the unit clause rule is applied, clause learning can also be incorporated in such circuit level DPLL-based solvers [35, 46]. Thus the results in this paper concerning the relative power of input-restricted clause learning DPLL hold for such circuit level approaches, too. Finally, we note that for instance [35] does not consider input-restricted branching but applies a top-down branching based on justification frontiers. The relative proof complexity theoretic power of the related top-down branching restrictions is analyzed in [28, 29].

4.2 Restricting branching in DPLL and CL to inputs

In structured application domains of SAT solvers, such as automated planning and bounded model checking of hardware and software, the problem at hand is based on a transition relation, where the behavior of the underlying system is dependent solely on the *input* of the system. In the Boolean circuit encoding $\mathcal{C}^\tau$ of such a structured problem, the input is represented by the set of input gates of the circuit, $\mathsf{inputs}(\mathcal{C})$. Since the values of the other gates in the circuit can be evaluated when all the gates in $\mathsf{inputs}(\mathcal{C})$ have values, branching in DPLL *with unit propagation* can be restricted to the variables associated with $\mathsf{inputs}(\mathcal{C})$ without losing completeness. Intuitively, the idea is that since the number of input gates $|\mathsf{inputs}(\mathcal{C})|$ is often much less than the total amount $|G|$ of gates in $\mathcal{C}$, the search space size is reduced from $2^{|G|}$ to $2^{|\mathsf{inputs}(\mathcal{C})|}$, where $|\mathsf{inputs}(\mathcal{C})| \ll |G|$.

By allowing branching in the DPLL and CL proof systems on input gates only, we arrive at the proof systems $\mathsf{DPLL}_{\mathsf{inputs}}$ and $\mathsf{CL}_{\mathsf{inputs}}$, respectively. From the view of proof complexity, however, in [29] a formal study on the effect of restricting branching in DPLL (without clause learning) to $\mathsf{inputs}(\mathcal{C})$ reveals that this weakens the proof system considerably.

Theorem 4 ([29]) $\mathsf{DPLL}_{\mathsf{inputs}}$ *cannot polynomially simulate* DPLL.

In the following section, we investigate the proof complexity theoretic effect of input-restricted branching in the context of *clause learning* DPLL-based SAT solving, which is posed as an open question in [29]. In Section 6 we complement this theoretical study by providing an experimental evaluation of the effect of input-restricted branching.

5 Restricted branching and proof complexity

We will now consider the relative proof complexity theoretic power of input-restricted and unrestricted branching CL and DPLL. This will result in the refined relative efficiency hierarchy of DPLL and CL shown in Fig. 4. An arrow without a slash from system S to S' means that S can polynomially simulate S', and with a slash that S cannot simulate S'. Arrows labeled with a $*$ are due to trivial subsumption. The new results, detailed in the following, are represented by dashed arrows. The missing arrows, disregarding those implied by the transitivity of the results, represent questions which are open to the best of our knowledge.

The main result of this paper is characterized by the following theorem.

Theorem 5 DPLL *and* $\text{CL--}_{\text{inputs}}$ (*with or without restarts*) *are incomparable.*

This is a direct corollary of the forthcoming Lemmas 1 and 3. Thus we get the following as a direct corollary.

Corollary 2 $\text{CL--}_{\text{inputs}}$ *with unlimited restarts cannot polynomially simulate* CL.

We now proceed by proving Theorem 5 in two parts. First we show by a simple argument why DPLL cannot simulate $\text{CL}_{\text{inputs}}$. We then discuss further the difference between $\text{CL}_{\text{inputs}}$ and $\text{DPLL}_{\text{inputs}}$ by exhibiting an example of a family of Boolean circuits on which $\text{CL}_{\text{inputs}}$ *can* simulate CL, while $\text{DPLL}_{\text{inputs}}$ *cannot* simulate DPLL. The motivation here is two-fold. On one hand, this shows the power of clause learning even when branching is restricted to inputs. On the other hand, the example gives an intuitive explanation of why the result in [29] on the power of $\text{DPLL}_{\text{inputs}}$ with respect to DPLL cannot be directly adapted for proving the analogous result for $\text{CL}_{\text{inputs}}$. Although $\text{CL}_{\text{inputs}}$ can simulate CL on this particular family of circuits, this is not the case in general for other families. After the example, we proceed by showing that in fact, $\text{CL--}_{\text{inputs}}$, even with conflict-driven backjumping and unlimited restarts, cannot even simulate DPLL. The proof relies on so called *redundant gates*, and applies known results on the very powerful *Extended Resolution* proof system [47].

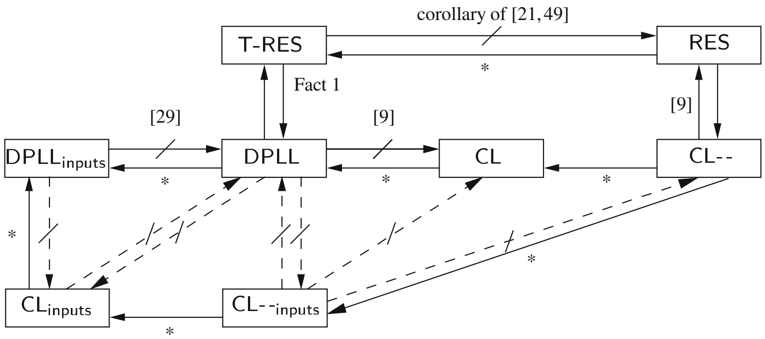


Fig. 4 A refined relative efficiency hierarchy for the proof systems considered in this work

5.1 DPLL cannot simulate $\mathsf{CL}_{\mathsf{inputs}}$

We now show that DPLL cannot simulate $\mathsf{CL}_{\mathsf{inputs}}$. This results from the fact that DPLL cannot simulate CL by additionally noticing that CL and $\mathsf{CL}_{\mathsf{inputs}}$ are equivalent when considering circuits representing CNF formulas.

Lemma 1 *There is an infinite family of constrained Boolean circuits for which* DPLL *has superpolynomially longer minimum proofs than* $\mathsf{CL}_{\mathsf{inputs}}$.

Proof Take any infinite family $\{F_n\}$ of CNF formulas that is a witness of Corollary 1 stating that DPLL cannot simulate CL. Define the family of Boolean circuits $\{\mathsf{circuit}(F) \mid F \in \{F_n\}\}$. The simplified CNF formula resulting from applying unit propagation to $\mathsf{cnf}(\mathsf{circuit}(F))$ is effectively the same as the simplified CNF formula resulting from applying unit propagation to F; especially, the OR-gate variables in $\mathsf{cnf}(\mathsf{circuit}(F))$ that represent the clauses in F are all assigned to **t**. Thus CL will only branch on the variables in $\mathsf{cnf}(\mathsf{circuit}(F))$ that are associated with the input gates of $\mathsf{circuit}(F)$ or their negations. Thus $\mathsf{CL}_{\mathsf{inputs}}$ can simulate CL on $\mathsf{cnf}(\mathsf{circuit}(F))$, and the claim follows by Corollary 1. □

As a direct corollary, we have

Corollary 3 *Neither* DPLL *nor* $\mathsf{DPLL}_{\mathsf{inputs}}$ *can polynomially simulate* $\mathsf{CL}_{\mathsf{inputs}}$.

Before considering whether $\mathsf{CL}_{\mathsf{inputs}}$ can simulate CL or DPLL, we next give a motivating example which illustrates why the results in [29] on the power of $\mathsf{DPLL}_{\mathsf{inputs}}$ with respect to DPLL cannot be directly adapted for proving the analogous result for $\mathsf{CL}_{\mathsf{inputs}}$.

5.2 A further motivating example

To highlight the strength of clause learning even when branching is restricted to input gates, we now give an example of a family, $\{\text{UNSAT-2PAR}_n\}$ where $n \geq 3$, of Boolean circuits on which $\mathsf{CL}_{\mathsf{inputs}}$ can simulate CL applying the 1-UIP learning scheme, although $\mathsf{DPLL}_{\mathsf{inputs}}$ cannot simulate DPLL on the family. The circuit

$$\text{UNSAT-2PAR}_n := \text{UNSAT} \cup \left\langle \text{PAR}_n^a \cup \text{PAR}_n^b, \emptyset \right\rangle$$

consists of two parts:

– the constant size circuit

$$\text{UNSAT} := \mathsf{circuit}\left(\{\{a,b\},\{a,\neg b\},\{\neg a,b\},\{\neg a,\neg b\}\}\right), \text{ and}$$

– two copies (for a and b, $\rho \in \{a,b\}$) of the circuit structure

$$\text{PAR}_n^\rho := \left\{\rho := \textsc{xor}\left(y_1^\rho, x_1^\rho\right)\right\} \cup \bigcup_{i=1}^{n-3} \left\{x_i^\rho := \textsc{xor}\left(y_{i+1}^\rho, x_{i+1}^\rho\right)\right\}$$
$$\cup \left\{x_{n-2}^\rho := \textsc{xor}\left(y_{n-1}^\rho, y_n^\rho\right)\right\}.$$

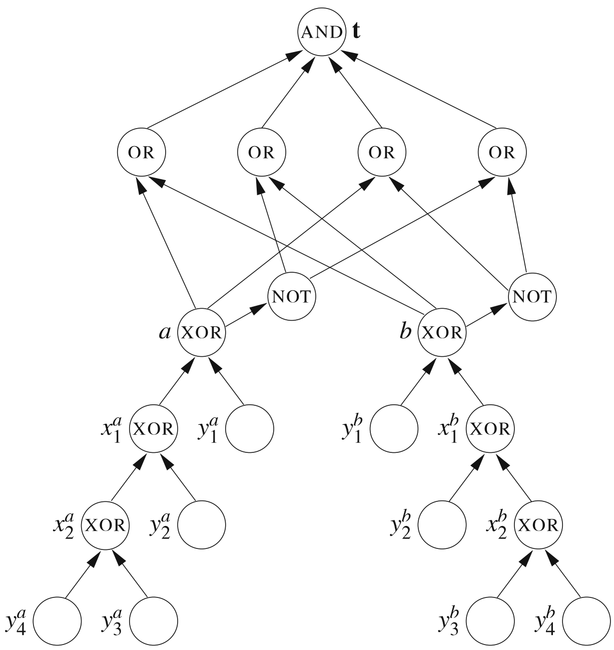


Fig. 5 The constrained Boolean circuit UNSAT-2PAR$_n$ for $n = 4$

Basically, PAR$_n^\rho$ computes the parity of the n input gates $y_1^\rho, \ldots, y_n^\rho$, evaluating to true if and only if an odd number of them are true.

The circuit UNSAT-2PAR$_4$ is shown in Fig. 5. Now, since unit propagation will result in a conflict in the UNSAT sub-circuit for any value of the gate a, UNSAT-2PAR$_n$ yields a trivial (constant length) proof in DPLL. It is also easy to see that minimum length proofs of UNSAT-2PAR$_n$ are exponential with respect to n in DPLL$_{\text{inputs}}$. This is because, due to the structure of PAR$_n^\rho$, in order to propagate a value for the gate a or b, DPLL$_{\text{inputs}}$ has to branch on all of the inputs in the corresponding PAR$_n^\rho$ sub-circuit. With the chronological backtracking process of DPLL this implies that minimum length DPLL$_{\text{inputs}}$ proofs of UNSAT-2PAR$_n$ are exponential with respect to n.

However, CL$_{\text{inputs}}$ can produce linear length proofs on the family. In the following we will say that CL (or DPLL) branches according to a sequence of assignments $(x_1 = v_1, x_2 = v_2, \ldots)$, if it always branches by assigning the value to the variable given by the next assignment in the sequence, i.e., we would first branch by assigning x_1 the value v_1, and so forth. Now, let CL$_{\text{inputs}}$ branch according to the sequence $(y_1^a = \mathbf{f}, \ldots, y_{n-1}^a = \mathbf{f})$. After this, unit propagation cannot still propagate a value on the gate a, any of the x_i^a gates, or any gate in the UNSAT sub-circuit. Then branch with $y_n^a = \mathbf{f}$. Now unit propagation sets values for all x_i^a gates without a conflict. The values for x_1^a and y_1^a propagate the value $\mathbf{f}$ for a, which then propagates a conflict at a gate in UNSAT. Notice that x_1^a and y_1^a are the *only* reasons for the value of a. In *any* conflict graph associated with the branching sequence $(y_1^a = \mathbf{f}, \ldots, y_n^a = \mathbf{f})$, $\neg a$ is a 1-UIP, and, furthermore, constitutes a reason for the conflict on its own. Hence CL$_{\text{inputs}}$ can learn as a unit clause the opposite value of a, and backjump to the decision level zero.

This opposite value will then propagate a conflict without branching, and $\mathsf{CL}_{\mathsf{inputs}}$ terminates.

It is interesting to notice how $\mathsf{CL}_{\mathsf{inputs}}$ can branch on $(y_1^a = \mathbf{f}, \ldots, y_n^a = \mathbf{f})$ and still avoid backtracking on these decisions since there is the *bottleneck* at gate a due to the construction of UNSAT-2PAR$_n$. This shows the strength of clause learning with conflict-driven backjumping—even with input-restricted branching—due to its ability to backjump over an exponential size search space by detecting small locally inconsistent sub-formulas. With this intuition, it is evident that the results in [29] on the power of $\mathsf{DPLL}_{\mathsf{inputs}}$ with respect to DPLL cannot be directly adapted for proving the analogous result for $\mathsf{CL}_{\mathsf{inputs}}$.

5.3 $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ cannot simulate DPLL

Although $\mathsf{CL}_{\mathsf{inputs}}$ can simulate CL on the {UNSAT-2PAR$_n$} family, this is generally not the case for other families. In fact, it turns out that $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ *cannot even simulate* DPLL, as detailed next.

We will apply the concept of *redundant gates in constrained Boolean circuits*.

Definition 3 A gate in a constrained Boolean circuit $\mathscr{C}^\tau$ is *redundant* if it is unconstrained and not a descendant of any constrained gate.

We will assume that circuits do not contain redundant input gates; such inputs can always be assigned an arbitrary truth value without affecting satisfiability. As shown next, when considering $\mathsf{CL}\text{--}_{\mathsf{inputs}}$, redundant gates cannot appear in conflict graphs. Intuitively, this is because redundant gates can only have a value due to unit propagation “upwards” (from child to parent) on the circuit structure in $\mathsf{CL}\text{--}_{\mathsf{inputs}}$; as they, or any of their parents, are not constrained by definition, they cannot cause a conflict or be a part of a unit propagation chain responsible for a conflict. As a consequence of this, redundant gates can never appear in conflict clauses derived by $\mathsf{CL}\text{--}_{\mathsf{inputs}}$.

Lemma 2 *Let $\mathscr{C}^\tau$ be an arbitrary constrained Boolean circuit. Considering $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ on input $\mathsf{cnf}(\mathscr{C}^\tau)$, redundant gates do not occur in any conflict graph at any stage of $\mathsf{CL}\text{--}_{\mathsf{inputs}}$. This holds whether or not restarts are allowed.*

Proof Take any constrained Boolean circuit $\mathscr{C}^\tau$. The stages in which $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ does not have a conflict are trivial. Now assume that the lemma holds at a stage where $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ has made m conflicts. Consider the stage producing the $(m+1)$th conflict and any conflict graph associated with the conflict. We next show that the conflict graph contains no redundant gates. Take any redundant gate g in $\mathscr{C}^\tau$. If it is not assigned, it cannot appear in the conflict graph. Now assume that g is assigned. Since g is redundant, it cannot be constrained by τ. Furthermore, g is not an input gate (by the assumption we made above), and thus g is assigned not because it was branched on. Therefore, g has been assigned by unit propagation. Now there are three cases.

– By the induction hypothesis, there are no known learned clauses containing redundant gates before the $(m+1)$th conflict, and therefore g is not assigned by unit propagation on a learned clause.

- The gate g is assigned because some of its children are assigned, i.e., by unit propagation on one of the clauses in $\mathsf{cnf}(\mathscr{C}^\tau)$ resulting from the equality $g \Leftrightarrow f(g_1, \ldots, g_n)$. Once g becomes assigned in this way, all these clauses become satisfied. Therefore, the value assigned to g by unit propagation could not have caused any of the children of g to be assigned.
- The gate g is assigned due to an assigned value on a parent g' of g, i.e., by unit propagation on one of the clauses in $\mathsf{cnf}(\mathscr{C}^\tau)$ resulting from the equality $g' \Leftrightarrow f(\ldots, g, \ldots)$. Since g is redundant, g' is also redundant. By the arguments above, the only way for g' to have been assigned in this situation is due to one of its parents' assigned value. Inductively, this leads to the fact that a redundant output gate o should have been assigned by unit propagation because one of o's parents has been assigned. This is a contradiction, since output gates have no parents. Therefore, the redundant gate g cannot be assigned because one of its parents is assigned.

Hence, the only reason for a redundant gate to be assigned is that some of its children are assigned. Furthermore, the value of an assigned redundant gate can only propagate values to its parents (which are also redundant). On the other hand, since redundant gates are not constrained by τ, g cannot act as the conflict variable in the conflict graph. Therefore, there cannot be any path from g to the conflict node in the implication graph which the conflict graph is based on. This proves that a redundant gate cannot occur in the conflict graph. □

Although redundant gates can be removed from any constrained Boolean circuit without affecting its satisfiability, they may have an effect on the length of shortest proofs. Cook [12] gives a way of introducing a polynomial number of clauses which can be interpreted as redundant gates to $\mathsf{circuit}(\mathrm{PHP}_n^{n+1})$ so that, contrarily to $\mathsf{circuit}(\mathrm{PHP}_n^{n+1})$, the extended circuit yields polynomial length proofs in RES. As a circuit structure, this *extension* is defined as $\mathrm{EXT}_n := \bigcup_{l=3}^{n+1} \mathrm{EXT}^l$, where

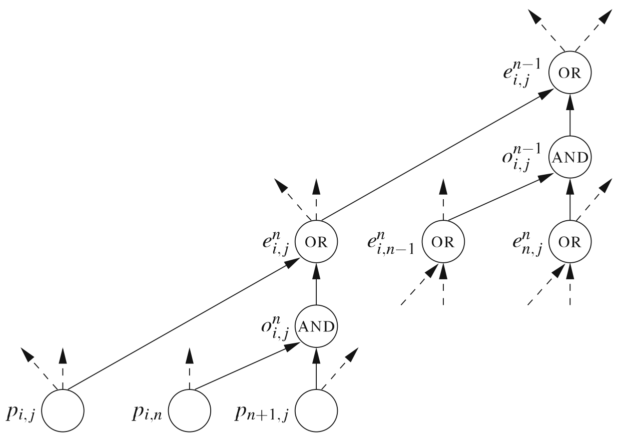
$$\mathrm{EXT}^l := \bigcup_{i=1}^{l-1} \bigcup_{j=1}^{l-2} \left\{ o_{i,j}^{l-1} := \textsc{and}\left(e_{i,l-1}^l, e_{l,j}^l\right),\ e_{i,j}^{l-1} := \textsc{or}\left(e_{i,j}^l, o_{i,j}^{l-1}\right) \right\},$$

and each $e_{i,j}^{n+1}$ is the gate $p_{i,j}$ in $\mathsf{circuit}(\mathrm{PHP}_n^{n+1})$. A part of EXT_n is illustrated in Fig. 6. The output gates of EXT_n are $e_{1,1}^2$ and $e_{2,1}^2, e_{3,2}^3, \ldots, e_{n,n-1}^n$.

Due to the result in [12], we immediately have a polynomial length RES proof $\pi = (C_1, \ldots, C_m = \emptyset)$ of the extended PHP_n^{n+1} formula $\mathsf{cnf}(\mathsf{circuit}(\mathrm{PHP}_n^{n+1}) \cup \langle \mathrm{EXT}_n, \emptyset \rangle)$. Intuitively, EXT^l allows reducing PHP_l^{l+1} to PHP_{l-1}^l with a polynomial number of resolution steps. However, since in [12] such a proof is not given explicitly, we include a detailed description of the proof in Appendix. For the following, what is most important is that such a short proof π exists, not really the actual details of π.[1] The details of π, along with EXT_n, are included here for the sake of concreteness and illustration.

[1] See Remark 6 in Section 5.4 for more details.

Fig. 6 Part of Cook's extension EXT_n to PHP_n^{n+1} as a circuit



Using the above-described polynomial length RES proof $\pi = (C_1, C_2, \ldots, C_m = \emptyset)$ for $\mathsf{cnf}(\mathsf{circuit}(\mathrm{PHP}_n^{n+1}) \cup \langle \mathrm{EXT}_n, \emptyset\rangle)$, we define the circuit construct

$$\mathrm{E}(\pi) := \bigcup_{i=1}^{m-1} \left\{g_{C_i} := \textsc{or}\left(g_1, \ldots, g_j, \hat{g}_{j+1}, \ldots, \hat{g}_k\right) \mid C_i = \left\{\tilde{g}_1, \ldots, \tilde{g}_j, \neg\tilde{g}_{j+1}, \ldots, \neg\tilde{g}_k\right\}\right\}$$

$$\cup \bigcup_{i=1}^{m-1} \left\{\hat{g} := \textsc{not}(g) \mid \tilde{g} \in \mathsf{vars}^-(C_i)\right\}.$$

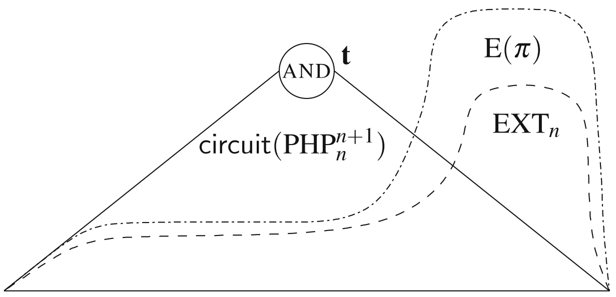
That is, each clause C_i in the RES proof π is simply represented as a corresponding OR-gate.

This allows a simple polynomial length DPLL proof of

$$\mathrm{EPHP}_n^{n+1} := \mathsf{circuit}(\mathrm{PHP}_n^{n+1}) \cup \langle \mathrm{EXT}_n, \emptyset\rangle \cup \langle \mathrm{E}(\pi), \emptyset\rangle,$$

while there is no polynomial length proof of EPHP_n^{n+1} in $\mathsf{CL}\text{-}\text{-}_{\mathsf{inputs}}$. Intuitively this is because $\mathrm{E}(\pi)$ allows DPLL to "verify" the resolution proof of PHP_n^{n+1} extended with EXT_n step-by-step, while $\mathsf{CL}\text{-}\text{-}_{\mathsf{inputs}}$ cannot make use of the redundant gates of EXT_n and $\mathrm{E}(\pi)$. For a high-level view of the structure of EPHP_n^{n+1}, see Fig. 7.

Fig. 7 High-level view of EPHP_n^{n+1}



Lemma 3 *For the infinite family* $\{\mathrm{EPHP}_n^{n+1}\}$ *of constrained Boolean circuits,* $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ *with unlimited restarts has superpolynomially longer minimum-length proofs than* DPLL.

Proof A polynomial length DPLL proof of EPHP_n^{n+1} is witnessed by the branching sequence $(g_{C_1} = \mathbf{f}, g_{C_2} = \mathbf{f}, \ldots, g_{C_{m-1}} = \mathbf{f})$, as detailed next. By induction on i, we will show that, if $g_{C_1} = \mathbf{t}, \ldots, g_{C_{i-1}} = \mathbf{t}$, then branching with $g_{C_i} = \mathbf{f}$ results in a conflict by unit propagation, and hence immediately sets $g_{C_i} = \mathbf{t}$.

The base case. The gate g_{C_1} represents the first clause C_1 in π, and thus C_1 must belong to $\mathsf{cnf}(\mathsf{circuit}(\mathrm{PHP}_n^{n+1}) \cup \langle \mathrm{EXT}_n, \emptyset\rangle)$. As C_1 is a result of applying the cnf translation to a gate g in $\mathsf{circuit}(\mathrm{PHP}_n^{n+1}) \cup \langle \mathrm{EXT}_n, \emptyset\rangle$ (which is part of EPHP_n^{n+1}), setting $g_{C_1} = \mathbf{f}$ will result in a conflict after unit propagation because the functional definition or the constraint of the gate g is violated. For example, if $g := \textsc{or}(g_1, g_2)$ and $C_1 = (\tilde{g} \vee \neg\tilde{g}_1)$, then $g_{C_1} := \textsc{or}(g, \hat{g}_1)$ with $\hat{g}_1 := \textsc{not}(g_1)$, and the assignment $g_{C_1} = \mathbf{f}$ will propagate $g = \mathbf{f}$ and $g_1 = \mathbf{t}$, violating the definition of g and thus resulting in a conflict.

Now assume as the induction hypothesis that we have $g_{C_{i'}} = \mathbf{t}$ for all $1 \le i' < i$. Next branch with $g_{C_i} = \mathbf{f}$. If the ith clause C_i in π belongs to $\mathsf{cnf}(\mathsf{circuit}(\mathrm{PHP}_n^{n+1}) \cup \langle \mathrm{EXT}_n, \emptyset\rangle)$, branching on $g_{C_i} = \mathbf{f}$ will result in a conflict after unit propagation as in the base case. Otherwise C_i has been derived from two clauses, $C_j = C'_j \cup \{\tilde{g}\}$ and $C_k = C'_k \cup \{\neg\tilde{g}\}$, in π for $1 \le j, k < i$, by resolving on the variable $\tilde{g}$. By the induction hypothesis we have $g_{C_j} = \mathbf{t}$ and $g_{C_k} = \mathbf{t}$. On the other hand, as $g_{C_i} = \mathbf{f}$, all the gates corresponding to the literals in $C'_j \cup C'_k$ are assigned to $\mathbf{f}$ by unit propagation, implying that unit propagation will assign both $g = \mathbf{t}$ and $g = \mathbf{f}$ as $g_{C_j} = g_{C_k} = \mathbf{t}$. Thus a conflict is reached, closing the branch $g_{C_i} = \mathbf{f}$, and $g_{C_i} = \mathbf{t}$ is set by backtracking.

Finally, since $C_m = \emptyset \in \pi$, there are unit clauses $C_j = \{\tilde{g}\}$ and $C_k = \{\neg\tilde{g}\}$ in π, where $1 \le j, k < m$. Without loss of generality, assume that $j < k$. By induction, at latest after branching with $g_{C_k} = \mathbf{f}$ and setting $g_{C_k} = \mathbf{t}$ by backtracking, we will have $g_{C_j} = g_{C_k} = \mathbf{t}$ in the branch, and thus both $g = \mathbf{t}$ and $g = \mathbf{f}$, a conflict. This closes the last branch, and we have a linear size DPLL proof of EPHP_n^{n+1}.

Now consider proofs of EPHP_n^{n+1} in $\mathsf{CL}\text{--}_{\mathsf{inputs}}$. The non-input gates in $\langle \mathrm{EXT}_n, \emptyset\rangle \cup \langle \mathrm{E}(\pi), \emptyset\rangle$ are all redundant in EPHP_n^{n+1}, and they cannot be part of a reason for any conflict in $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ (Lemma 2). Thus any $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ proof of EPHP_n^{n+1} contains a $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ proof of PHP_n^{n+1}, which cannot be of polynomial length (Theorems 1 and 3). □

Theorem 5 now follows directly from Lemmas 1 and 3.

5.4 Additional remarks

Closely related to Lemma 3 and the applied construction EPHP_n^{n+1}, we make the following additional remarks.

1. Due to the fact that redundant gates do not occur in *any* conflict graph of $\mathsf{CL}\text{--}_{\mathsf{inputs}}$, Lemma 3 covers all clause learning schemes based on conflict cuts, including, for example, schemes which learn *multiple clauses* at each conflict [39]. Additionally, conflict clause forgetting schemes, which are applied in typical clause learning solvers such as [18], do not affect this result.

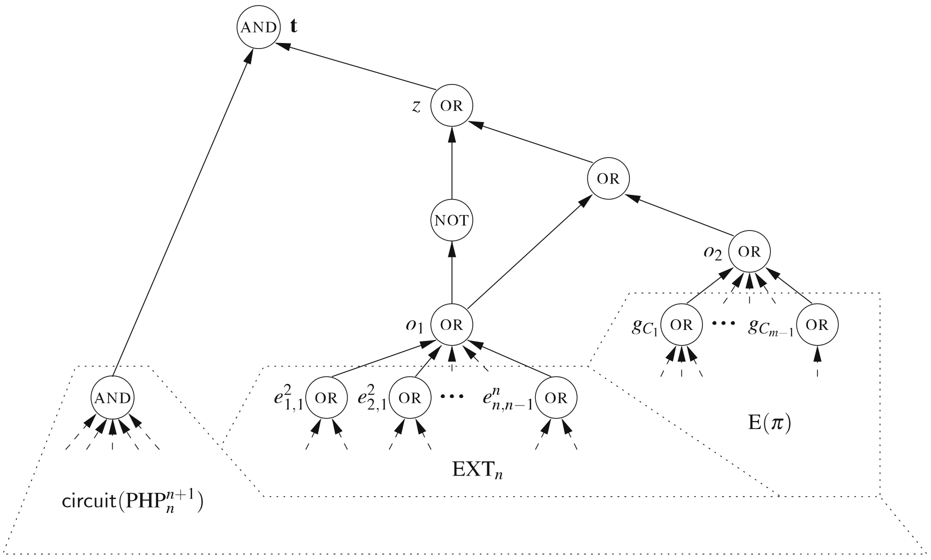


Fig. 8 Local change to the EPHP$^{n+1}_n$ circuit for removing redundancy of gates in E(π) and EXT$_n$

2. We use redundant gates in the EPHP$^{n+1}_n$ construction for simplicity of the proof of Lemma 3; by a simple modification of EPHP$^{n+1}_n$ one can construct as a witness for Lemma 3 a constrained circuit with no redundant gates and a single output as the only constrained gate. The basic idea, illustrated in Fig. 8, is to make a small local change to the EPHP$^{n+1}_n$ circuit construct. In more detail, introduce the OR-gate o_1 over the output gates $e^2_{1,1}, e^2_{2,1}, \ldots, e^n_{n,n-1}$ in EXT$_n$. Similarly, introduce the OR-gate o_2 over the output gates $g_{C_1}, \ldots, g_{C_{m-1}}$ in E(π). Now, introduce an OR-gate over o_1 and o_2. Then, introduce a gate z that is the OR of this gate and a new gate NOT(o_1). Finally, constrain the AND of this gate and the output gate of the unconstrained version of circuit(PHP$^{n+1}_n$) to **t**. The resulting circuit family can be used in proving Lemma 3 as the values propagated to the non-input gates in EXT$_n$ and E(π) cannot be part of any conflict graph in CL--$_{\text{inputs}}$. This is because the gate z always evaluates to **t**; it corresponds to a tautology of form $\neg a \vee (a \vee b)$ and thus effectively makes EXT$_n$ and E(π) redundant.
3. Since redundant gates can be removed from constrained Boolean circuits without affecting the existence of satisfying assignments, such gates are typically removed in practice before the CNF translation and SAT solving by using the so-called *cone-of-influence reduction* [31]. However, applying the cone-of-influence reduction can have a drastic negative effect on minimum length proofs: if one applies the cone-of-influence reduction to the circuit family EPHP$^{n+1}_n$, one obtains the family PHP$^{n+1}_n$ for which CL-- does not have polynomial length proofs although the much weaker system DPLL has short proofs for the original family EPHP$^{n+1}_n$ (as shown in the proof of Lemma 3).
4. It is interesting to notice that DPLL solvers with full one-step lookahead can detect the small proofs of EPHP$^{n+1}_n$ witnessed by the branching sequence

($g_{C_1} = \mathbf{f}, g_{C_2} = \mathbf{f}, \ldots, g_{C_{m-1}} = \mathbf{f}$). In particular, for each i, lookahead on $g_{C_i} = \mathbf{f}$ when having $g_{C_j} = \mathbf{t}$ for all $j < i$ in the branch will result in an immediate conflict using unit propagation, as detailed in the proof of Lemma 3.

5. The Cook's extension (a variant of EXT_n) presented in [12] is motivated by investigations into the power of the *Extended Resolution proof system* defined by Tseitin [47]. Extended Resolution is the result of adding an *extension rule* to RES, which allows for iteratively adding *definitions* of the form $x \Leftrightarrow l_1 \wedge l_2$ (or, as a set of clauses, $\{\{x, \neg l_1, \neg l_2\}, \{\neg x, l_1\}, \{\neg x, l_2\}\}$) to the CNF formula, where x is a new variable and l_1, l_2 are literals in the current formula. This is equivalent to adding a redundant binary AND gate of the literals l_1, l_2 to a constrained Boolean circuit. Notably, it is known that Extended Resolution is among the most powerful proof systems, and can simulate, e.g., *Frege systems* (see [34] for more details).
6. Instead of the pigeon-hole problem PHP_n^{n+1}, Cook's extension EXT_n to it, and the resolution proof π of their combination, one could use any CNF formula F that (i) does not have a polynomial length resolution proof but (ii) has a polynomial length extended resolution proof to prove a result similar to Lemma 3. That is, for such formula F, DPLL has a polynomial length proof of $\mathsf{circuit}(F) \cup \langle \mathrm{EXT}_F, \emptyset\rangle \cup \langle \mathrm{E}(\pi_F), \emptyset\rangle$ while $\mathsf{CL\text{-}\text{-}}_{\mathsf{inputs}}$ does not, where EXT_F is the polynomial length extension of F and π_F is a polynomial length resolution proof of $\mathsf{cnf}(\mathsf{circuit}(F) \cup \langle \mathrm{EXT}_F, \emptyset\rangle)$.
7. The additional extension $\mathrm{E}(\pi)$ applied above is motivated by a similar construction which can be used for simulating *Frege proofs* with their tree-like variants (see [34, Chapter 4]).

6 Experiments

We evaluate the effect of restricting branching to input variables on the functionality of modern clause learning solver techniques. The set of benchmarks[2] used in the experiments consists of instances from various application domains, for which Boolean circuits offer a natural representation form: super-scalar processor verification [50], integer factorization based on hardware multipliers [43], equivalence checking of hardware multipliers [26], bounded model checking (BMC) for deadlocks in asynchronous parallel systems represented as labelled transition systems (LTS) [32], and linear temporal logic (LTL) BMC of finite state systems with a compact encoding [36]. We use standard PCs with 2-GHz AMD 3200+ processors and two gigabytes of memory running Linux, and apply a timeout of one hour and a memory limit of one gigabyte to each SAT solver execution.

For solving the Boolean circuit instances, we apply BCMinisat[3] (version 0.26), which we have modified in order to restrict branching to input variables. BCMinisat is

[2]The set of Boolean circuit benchmarks is available at http://www.tcs.hut.fi/~mjj/benchmarks/.

[3]Part of the BCTools package, http://www.tcs.hut.fi/~tjunttil/bcsat/.

a Boolean circuit front-end for the successful clause learning SAT solver Minisat [18] (version 1.14). BCMinisat accepts as input Boolean circuits with various Boolean functions allowed as gate types, performs circuit-level preprocessing, including Boolean propagation, substructure sharing, and cone-of-influence reductions to the circuit, normalizing the circuit into a form which can be translated into CNF applying a standard translation in the style of cnf defined in Table 1. BCMinisat feeds the resulting CNF translation and the input-restriction to Minisat, which then solves the CNF. For each circuit, we obtain 15 CNF instances by permuting the CNF variable numbering.

Minisat implements 1-UIP clause learning. After each conflict the heuristic value of each variable on the conflict side and in the conflict clause is incremented by one, and the values of all variables are decremented by 5%. To avoid hindering efficiency by learning massive amounts of clauses, the solver also uses a scheme for forgetting learned clauses that have not occurred on the conflict side in recent conflicts.

6.1 Results

Table 2 gives the minimum, median, and maximum number of decisions for BCMinisat and input-restricted BCMinisat ($\text{BCMinisat}_{\mathsf{inputs}}$) for each benchmark instance. For the instances based on hardware multiplication designs, for which the number of unassigned input variables is 2% or less out of all unassigned variables, $\text{BCMinisat}_{\mathsf{inputs}}$ shows an advantage over BCMinisat with respect to the number of decisions. However, for the hardware verification and BMC instances, the overall performance of $\text{BCMinisat}_{\mathsf{inputs}}$ is much worse, with timeouts on all verification and half of the LTL BMC instances. The possible gains of applying input-restricted branching seem to correlate with a very low relative number of input variables. On the equivalence checking instances, we notice that the number of decision for $\text{BCMinisat}_{\mathsf{inputs}}$ *is more than the brute-force upper bound*, e.g., for `eq-test.atree.braun.10` around $1.4 - 1.8 \times 10^6$, compared to the brute-force bound $2^{20} \approx 1.0 \times 10^6$. Considering that we are using a state-of-the-art clause learning solver, this surprising result is likely due to conflict clause forgetting;[4] when forgetting a conflict clause C, the solver may have to re-examine the search space characterised as unsatisfiable by C. Figure 9 gives a cumulative plot of the number of solved instances, showing a drastic decrease in performance for the input-restriction.

The effect of input-restriction varies depending on whether unsatisfiable or satisfiable instances are considered (Fig. 10). For the unsatisfiable instances the plot correlates well with Corollary 2, with timed out runs on the horizontal line. For satisfiable instances, there seems to be no clear winner, although when selecting from the relative small set of input variables, the probability of choosing a satisfying assignment is intuitively greater.

We also observe that the VSIDS branching heuristics [40] applied in Minisat might not work as intended with the input-restriction. The number of unbranchable variables which have better heuristic values than the best branchable variable can be high per decision (median of averages: **ud** in Table 2), e.g., for `eq-test.atree.`

[4]For more evidence corroborating this claim, see [30].

Table 2 Minimum (*min*), median (*med*), and maximum (*max*) of number of decisions for BCMinisat and BCMinisat$_{\text{inputs}}$, with number of timeouts in parenthesis

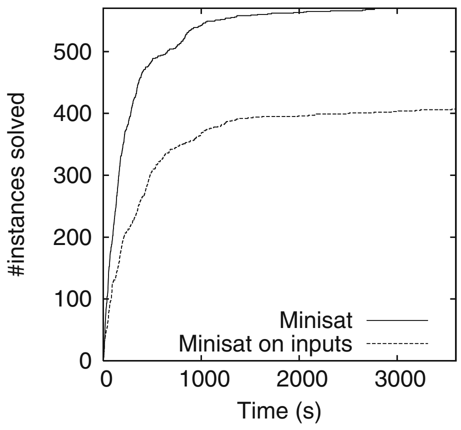
Instance	sat	#inputs	Number of decisions							
			BCMinisat			BCMinisat$_{\text{inputs}}$				
			min	med	max	min	med	max	ud	bb
Super-scalar processor verification										
`fvp.2.0.3pipe.1`	No	186 (8.2)	*61531*	*384386*	*1225134*	– (15)	– (15)	– (15)	–	–
`fvp.2.0.3pipe_2_ooo.1`	No	305 (11.7)	*75962*	*184798*	*426489*	– (15)	– (15)	– (15)	–	–
`fvp.2.0.4pipe_1_ooo.1`	No	544 (10.4)	*188992*	*209048*	*271982*	– (15)	– (15)	– (15)	–	–
`fvp.2.0.4pipe_2_ooo.1`	No	547 (9.8)	*1033607*	*2094617*	*5241781*	– (15)	– (15)	– (15)	–	–
`fvp.2.0.5pipe_1_ooo.1`	No	845 (8.9)	*336281*	*746231*	*1838599*	– (15)	– (15)	– (15)	–	–
Equivalence checking hardware multipliers										
`eq-test.atree.braun.8`	No	16 (2.3)	180449	285665	339805	*65785*	*73834*	*82372*	88.5	0.02
`eq-test.atree.braun.9`	No	18 (2.0)	898917	1055511	1317785	*323688*	*385398*	*389890*	106.6	0.02
`eq-test.atree.braun.10`	No	20 (1.8)	3755375	4540598	5089443	*1428957*	*1590390*	*1787295*	127.9	0.01
Integer factorisation										
`atree.sat.34.0`	Yes	60 (0.6)	156733	228792	761620	*24820*	*208880*	*277896*	21.9	0.04
`atree.sat.36.50`	Yes	64 (0.6)	*251218*	721474	937152	316590	*571533*	*788762*	18.4	0.04
`atree.sat.38.100`	Yes	68 (0.6)	284980	1095192	– (1)	*190330*	*498092*	*1082729*	–	–
`atree.unsat.32.0`	No	57 (0.7)	141419	163508	180973	*123502*	*138797*	*162546*	15.3	0.04
`atree.unsat.34.50`	No	60 (0.6)	248371	287351	404418	*223130*	*244382*	*301464*	18.0	0.04
`atree.unsat.36.100`	No	64 (0.6)	527237	623889	915810	*431576*	*480469*	*578331*	19.4	0.03
`braun.sat.32.0`	Yes	61 (2.2)	27480	82122	140150	*5675*	*81269*	*135093*	25.6	0.05
`braun.sat.34.50`	Yes	65 (2.1)	*30717*	152224	353464	43924	*110614*	*223306*	25.3	0.05
`braun.sat.36.100`	Yes	69 (2.0)	129771	447716	*589449*	*86134*	*374884*	752645	19.4	0.05
`braun.unsat.32.0`	No	60 (2.2)	107617	122550	156004	*96894*	*119437*	*150121*	10.4	0.06
`braun.unsat.34.50`	No	64 (2.0)	215624	263845	341855	*213199*	*258446*	*316819*	9.1	0.06
`braun.unsat.36.100`	No	68 (1.9)	*514725*	*623671*	807610	533575	640111	*674470*	8.9	0.06

Table 2 (continued)

Instance	sat	#inputs	Number of decisions							
			BCMinisat			BCMinisat$_{\text{inputs}}$				
			min	med	max	min	med	max	ud	bb
BMC for deadlocks in LTSs										
`dp_12.i.k10`	No	480 (16.0)	*513935*	*639756*	*987595*	2497570	– (10)	– (10)	–	–
`key_4.p.k28`	No	967 (10.9)	*121552*	*147063*	*169386*	138361	184875	220107	3.7	0.53
`key_4.p.k37`	Yes	1507 (9.8)	56784	*321552*	*1549271*	*7574*	663152	– (1)	–	–
`key_5.p.k29`	No	1212 (10.7)	*193139*	*223867*	*310207*	230844	343255	405686	3.9	0.54
`key_5.p.k37`	Yes	1796 (9.8)	104496	*421324*	*1540174*	*19027*	1041807	– (3)	–	–
`mmgt_4.i.k15`	No	456 (10.9)	*210288*	*287599*	*457009*	582998	1105986	2170048	4.2	0.41
`q_1.i.k18`	No	566 (13.1)	*168156*	*353421*	*507246*	375493	929019	1349785	3.7	0.49
LTL BMC by linear encoding										
`1394-4-3.p1neg.k10`	No	1845 (5.6)	141822	155295	164900	*138468*	*148545*	*156839*	6.6	0.34
`1394-4-3.p1neg.k11`	Yes	2023 (5.5)	72988	128708	203647	*34619*	*55575*	*189434*	9.0	0.32
`1394-5-2.p0neg.k13`	No	1940 (5.0)	*125840*	*143928*	*158320*	146144	156527	186468	6.7	0.32
`brp.ptimonegnv.k23`	No	461 (6.7)	*106338*	*130577*	*259025*	193839	302930	356313	4.1	0.28
`brp.ptimonegnv.k24`	Yes	481 (6.7)	43013	96775	*162114*	*13699*	*74907*	260481	5.5	0.27
`csmacd.p0.k16`	No	1794 (2.9)	*229192*	*316082*	*376280*	269520	341751	381248	4.9	0.28
`dme3.ptimo.k61`	No	6375 (26.3)	*314659*	*549686*	*1658757*	– (15)	– (15)	– (15)	–	–
`dme3.ptimo.k62`	Yes	6506 (26.3)	*427100*	*688505*	*1545603*	– (15)	– (15)	– (15)	–	–
`dme3.ptimonegnv.k58`	No	5982 (26.3)	*324770*	*568864*	*962967*	– (15)	– (15)	– (15)	–	–
`dme3.ptimonegnv.k59`	Yes	6113 (26.3)	*303921*	*480073*	*1136938*	– (15)	– (15)	– (15)	–	–
`dme5.ptimo.k65`	No	10750 (26.8)	*497190*	*735741*	*1839619*	– (15)	– (15)	– (15)	–	–

For each instance, the smaller of the two min, med, and max values is emphasized
The *sat* column gives the satisfiability of the instance, and #inputs gives the number of unassigned input variables in the CNF translation (percentage in parentheses).
For **ud** and **bb**, see the text body

Fig. 9 Comparison of input-restricted branching and unrestricted Minisat: cumulative number of solved instances



braun.10 on the average there are, per decision, over 100 unbranchable variables with better heuristic scores than the best branchable one. From another point of view, the fraction of increments on branchable variables from the number of all increments to heuristic values during search can be in some cases even as low as 1% (median: **bb** in Table 2)—running the risk of VSIDS degenerating into a random heuristic. These observations imply that in order to incorporate branching restrictions in clause learning solvers, the restriction itself should be taken into account in developing suitable heuristics and learning schemes.

As a final remark, we refer to [30] for a more in-depth experimental investigation into the effects of restricted branching—not limited to the input-restriction and hence extending the experimental evidence provided here—on the efficiency of clause learning solvers.

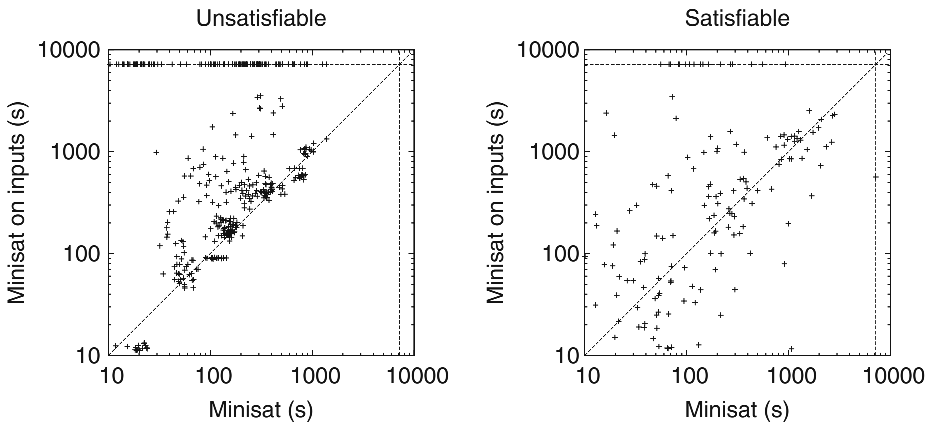


Fig. 10 Comparison of input-restricted branching and unrestricted Minisat as scatter plots: running times on unsatisfiable (*left*) and satisfiable (*right*) instances

7 Conclusions

We investigate the effect of restricting branching in clause learning SAT solving on the efficiency of the underlying inference system from the view of proof complexity. It is known that the unrestricted version of the considered variant of clause learning can efficiently simulate general resolution, being thus very powerful compared to the basic DPLL (with *no clause learning*). However, we show the surprising result that input-restricted clause learning cannot even simulate the basic DPLL. This implies that all implementations of clause learning, even with optimal heuristics, have the potential of suffering a notable efficiency decrease if branching is restricted to input variables. The experimental evidence shows that by restricting branching the robustness of SAT solvers can decrease, and that input-restricted branching does not go well with clause learning based heuristics of modern solvers.

Acknowledgements We thank Ilkka Niemelä for numerous discussions on the topic of this work, and Emilia Oikarinen for help on the resolution proof construction in Appendix.

Appendix Polynomial length RES proof of EPHP$_n^{n+1}$

The RES proof consists of four components, out of which the first three will be applied iteratively in a level-wise fashion from $l = n+1$ to $l = 3$. The intuitive idea is that at level l we will derive PHP$_{l-2}^{l-1}$ from PHP$_{l-1}^{l}$ and EXT^l in a polynomial number of resolution steps.

1. Resolve on the gates $o_{i,j}^{l-1}$, where $i = 1, \dots, l+1$ and $j = 1, \dots, l$, using the clauses in the CNF translation of $e_{i,j}^{l-1} := \text{OR}(e_{i,j}^{l}, o_{i,j}^{l-1})$ and $o_{i,j}^{l-1} := \text{AND}(e_{i,l-1}^{l}, e_{l,j}^{l})$.
2. Derive the long clause $\{e_{i,1}^{l-1}, \dots, e_{i,l-2}^{l-1}\}$ from $\{e_{i,1}^{l}, \dots, e_{i,l-1}^{l}\}$ for each $i = 1, \dots, l-1$.
3. Derive the short clauses of the form $\{\neg e_{i,k}^{l-1}, \neg e_{j,k}^{l-1}\}$ for $1 \le i, j \le l-1$ and $1 \le k \le l-2$.
4. After iterating steps 1-3 from $l = n+1$ down to $l = 3$, derive the empty clause in two step from the clauses in PHP$_1^2$.

We will describe these steps now in more detail.

1. For each $e_{i,j}^{l-1} := \text{OR}(e_{i,j}^{l}, o_{i,j}^{l-1})$ we have the clauses

$$\left\{\neg e_{i,j}^{l-1}, e_{i,j}^{l}, o_{i,j}^{l-1}\right\}, \left\{e_{i,j}^{l-1}, \neg e_{i,j}^{l}\right\}, \left\{e_{i,j}^{l-1}, \neg o_{i,j}^{l-1}\right\},$$

   and for each $o_{i,j}^{l-1} := \text{AND}(e_{i,l-1}^{l}, e_{l,j}^{l})$ the clauses

$$\left\{o_{i,j}^{l-1}, \neg e_{i,l-1}^{l}, \neg e_{l,j}^{l}\right\}, \left\{\neg o_{i,j}^{l-1}, e_{i,l-1}^{l}\right\}, \left\{\neg o_{i,j}^{l-1}, e_{l,j}^{l}\right\}.$$

   In particular, when resolving on the gate $o_{i,j}^{l-1}$, we obtain from these clauses the clauses

$$\left\{\neg e_{i,j}^{l-1}, e_{i,j}^{l}, e_{i,l-1}^{l}\right\}, \left\{\neg e_{i,j}^{l-1}, e_{i,j}^{l}, e_{l,j}^{l}\right\}, \left\{e_{i,j}^{l-1}, \neg e_{i,l-1}^{l}, \neg e_{l,j}^{l}\right\}.$$

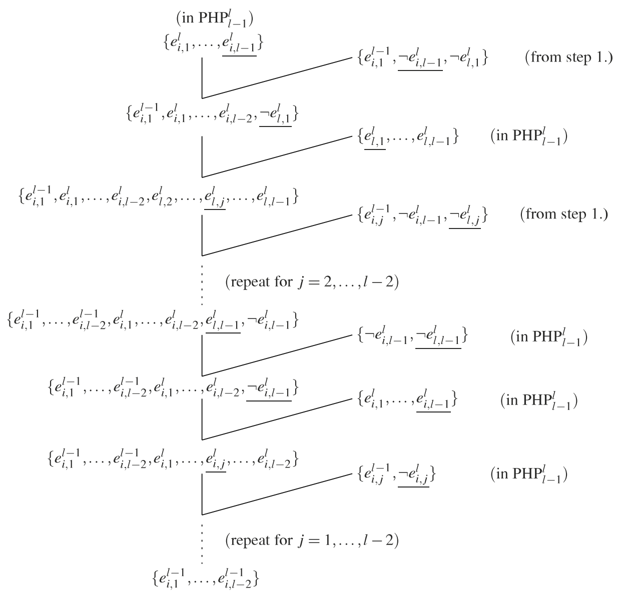

(in PHP^l_{l-1})
$\{e^l_{i,1},\dots,\underline{e^l_{i,l-1}}\}$
$\{e^{l-1}_{i,1},\underline{\neg e^l_{i,l-1}},\neg e^l_{l,1}\}$
(from step 1.)
$\{e^{l-1}_{i,1},e^l_{i,1},\dots,e^l_{i,l-2},\underline{\neg e^l_{l,1}}\}$
$\{\underline{e^l_{l,1}},\dots,e^l_{l,l-1}\}$
(in PHP^l_{l-1})
$\{e^{l-1}_{i,1},e^l_{i,1},\dots,e^l_{i,l-2},e^l_{l,2},\dots,\underline{e^l_{l,j}},\dots,e^l_{l,l-1}\}$
$\{e^{l-1}_{i,j},\neg e^l_{i,l-1},\underline{\neg e^l_{l,j}}\}$
(from step 1.)
(repeat for $j=2,\dots,l-2$)
$\{e^{l-1}_{i,1},\dots,e^{l-1}_{i,l-2},e^l_{i,1},\dots,e^l_{i,l-2},\underline{e^l_{l,l-1}},\neg e^l_{i,l-1}\}$
$\{\neg e^l_{i,l-1},\underline{\neg e^l_{l,l-1}}\}$
(in PHP^l_{l-1})
$\{e^{l-1}_{i,1},\dots,e^{l-1}_{i,l-2},e^l_{i,1},\dots,e^l_{i,l-2},\underline{\neg e^l_{i,l-1}}\}$
$\{e^l_{i,1},\dots,\underline{e^l_{i,l-1}}\}$
(in PHP^l_{l-1})
$\{e^{l-1}_{i,1},\dots,e^{l-1}_{i,l-2},e^l_{i,1},\dots,\underline{e^l_{i,j}},\dots,e^l_{i,l-2}\}$
$\{e^{l-1}_{i,j},\underline{\neg e^l_{i,j}}\}$
(in PHP^l_{l-1})
(repeat for $j=1,\dots,l-2$)
$\{e^{l-1}_{i,1},\dots,e^{l-1}_{i,l-2}\}$

Fig. 11 How to derive $\{e^{l-1}_{i,1},\dots,e^{l-1}_{i,l-2}\}$ in a polynomial number of resolution steps using Cook's extension for PHP^{n+1}_n

2. The derivation is described in Fig. 11. Notice that, at each step, the variable resolved upon is underlined. Recall that $\{e^{n+1}_{i,1},\dots,e^{n+1}_{i,n}\}$ is the clause $\{p_{i,1},\dots,p_{i,n}\}$ in PHP^{n+1}_n.
3. Figure 12 shows how to derive the clauses of the form $\{\neg e^{l-1}_{i,k},\neg e^{l-1}_{j,k}\}$.
4. By recursively applying the derivations in Figs. 11 and 12 from $l=n+1$ to $l=3$, one can thus derive the clauses $\{e^2_{1,1}\}$, $\{e^2_{2,1}\}$, and $\{\neg e^2_{1,1},\neg e^2_{2,1}\}$. Finally, the empty clause can be derived from these clauses with two resolution steps.

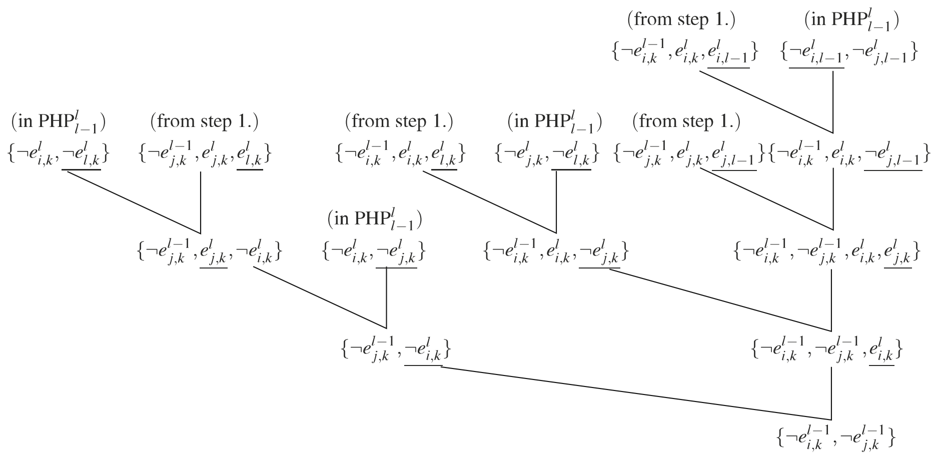

(from step 1.)
$\{\neg e^{l-1}_{i,k},e^l_{i,k},\underline{e^l_{i,l-1}}\}$
(in PHP^l_{l-1})
$\{\underline{\neg e^l_{i,l-1}},\neg e^l_{j,l-1}\}$
(in PHP^l_{l-1})
$\{\neg e^l_{i,k},\underline{\neg e^l_{l,k}}\}$
(from step 1.)
$\{\neg e^{l-1}_{j,k},e^l_{j,k},\underline{e^l_{l,k}}\}$
(from step 1.)
$\{\neg e^{l-1}_{i,k},e^l_{i,k},\underline{e^l_{l,k}}\}$
(in PHP^l_{l-1})
$\{\neg e^l_{j,k},\underline{\neg e^l_{l,k}}\}$
(from step 1.)
$\{\neg e^{l-1}_{j,k},e^l_{j,k},\underline{e^l_{j,l-1}}\}$
$\{\neg e^{l-1}_{i,k},e^l_{i,k},\underline{\neg e^l_{j,l-1}}\}$
$\{\neg e^{l-1}_{j,k},\underline{e^l_{j,k}},\neg e^l_{i,k}\}$
(in PHP^l_{l-1})
$\{\neg e^l_{i,k},\underline{\neg e^l_{j,k}}\}$
$\{\neg e^{l-1}_{i,k},e^l_{i,k},\underline{\neg e^l_{j,k}}\}$
$\{\neg e^{l-1}_{i,k},\neg e^{l-1}_{j,k},e^l_{i,k},\underline{e^l_{j,k}}\}$
$\{\neg e^{l-1}_{j,k},\underline{\neg e^l_{i,k}}\}$
$\{\neg e^{l-1}_{i,k},\neg e^{l-1}_{j,k},\underline{e^l_{i,k}}\}$
$\{\neg e^{l-1}_{i,k},\neg e^{l-1}_{j,k}\}$

Fig. 12 How to derive $\{\neg e^{l-1}_{i,k},\neg e^{l-1}_{j,k}\}$ in a polynomial number of steps using Cook's extension for PHP^{n+1}_n

However, one can see that derived clauses in each PHP^l_{l-1} are used multiple times in the RES proof. For example, for each l, the clause $\{e^l_{l,1}, \ldots, e^l_{l,l-1}\}$ is used in the order of l times in the derivation shown in Fig. 11. Hence the end result is not a T-RES proof.

References

1. Achlioptas, D., Beame, P., & Molloy, M. (2004). Exponential bounds for DPLL below the satisfiability threshold. In J. I. Munro (Ed.), *Proceedings of the 15th annual ACM-SIAM symposium on discrete algorithms (SODA'04)* (pp. 139–140). Philadelphia: SIAM.
2. Achlioptas, D., Beame, P., & Molloy, M. S. O. (2004). A sharp threshold in proof complexity yields lower bounds for satisfiability search. *Journal of Computer and System Sciences, 68*(2), 238–268.
3. Alekhnovich, M. (2004). Mutilated chessboard problem is exponentially hard for resolution. *Theoretical Computer Science, 310*(1–3), 513–525.
4. Alekhnovich, M., Hirsch, E. A., & Itsykson, D. (2005). Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. *Journal of Automated Reasoning, 35*(1–3), 51–72.
5. Alekhnovich, M., Johannsen, J., Pitassi, T., & Urquhart, A. (2002). An exponential separation between regular and general resolution. In *Proceedings on 34th annual ACM symposium on theory of computing (STOC'02)* (pp. 448–456). New York: ACM.
6. Beame, P., Culberson, J. C., Mitchell, D. G., & Moore, C. (2005). The resolution complexity of random graph *k*-colorability. *Discrete Applied Mathematics, 153*(1–3), 25–47.
7. Beame, P., Impagliazzo, R., & Sabharwal, A. (2007). The resolution complexity of independent sets and vertex covers in random graphs. *Computational Complexity, 16*(3), 245–297.
8. Beame, P., Karp, R. M., Pitassi, T., & Saks, M. E. (2002). The efficiency of resolution and Davis–Putnam procedures. *SIAM Journal on Computing, 31*(4), 1048–1075.
9. Beame, P., Kautz, H. A., & Sabharwal, A. (2004). Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research, 22*, 319–351.
10. Biere, A., Cimatti, A., Clarke, E. M., Fujita, M., & Zhu, Y. (1999). Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th conference on design automation (DAC'99)* (pp. 317–320). New York: ACM.
11. Chvátal, V., & Szemerédi, E. (1988). Many hard examples for resolution. *Journal of the ACM, 35*(4), 759–768.
12. Cook, S. A. (1976). A short proof of the pigeon hole principle using extended resolution. *SIGACT News, 8*(4), 28–32.
13. Cook, S. A., & Reckhow, R. A. (1979) The relative efficiency of propositional proof systems. *Journal of Symbolic Logic, 44*(1), 36–50.
14. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., et al. (2001). Benefits of bounded model checking at an industrial setting. In G. Berry, H. Comon, & A. Finkel (Eds.), *Proceedings of the 13th international conference on computer aided verification (CAV'01). Lecture notes in computer science* (Vol. 2102, pp. 436–453). New York: Springer.
15. Dantchev, S., & Riis, S. (2001). "Planar" tautologies hard for resolution. In *Proceedings of the 42nd IEEE symposium on foundations of computer science (FOCS'01)* (pp. 220–229). Los Alamitos: IEEE Computer Society.
16. Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem proving. *Communications of the ACM, 5*(7), 394–397.
17. Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the ACM, 7*(3), 201–215.
18. Eén, N., & Sörensson, N. (2004). An extensible SAT-solver. In E. Giunchiglia & A. Tacchella (Eds.), *Revised selected papers of the 6th international conference on theory and applications of satisfiability testing (SAT'03). Lecture notes in computer science* (Vol. 2919, pp. 502–518). New York: Springer.
19. Giunchiglia, E., Maratea, M., & Tacchella, A. (2002). Dependent and independent variables in propositional satisfiability. In S. Flesca, S. Greco, N. Leone, & G. Ianni (Eds.), *Proceedings of*

the European conference on logics in artificial intelligence JELIA'02. Lecture notes in artificial intelligence (Vol. 2424, pp. 296–307). New York: Springer.
20. Giunchiglia, E., Massarotto, A., & Sebastiani, R. (1998). Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In B. B. C. Rich, J. Mostow, & R. Uthurusamy (Eds.), *Proceedings of the 15th national conference on artificial intelligence* (*AAAI'98*) (pp. 948–953). Menlo Park: AAAI.
21. Goerdt, A. (1993). Regular resolution versus unrestricted resolution. *SIAM Journal on Computing, 22*(4), 661–683.
22. Goldberg, E., & Novikov, Y. (2002). Berkmin: A fast and robust SAT-solver. In *Proceedings of the 2002 design, automation and test in Europe conference (DATE'02)* (pp. 142–149). Los Alamitos: IEEE Computer Society.
23. Gomes, C. P., Selman, B., & Kautz, H. A. (1998). Boosting combinatorial search through randomization. In B. B. C. Rich, J. Mostow, & R. Uthurusamy (Eds.), *Proceedings of the 15th national conference on artificial intelligence (AAAI'98)* (pp. 431–437). Menlo Park: AAAI.
24. Haken, A. (1985). The intractability of resolution. *Theoretical Computer Science, 39*(2–3), 297–308.
25. Huang, J. (2007). The effect of restarts on the efficiency of clause learning. In M. M. Veloso (Ed.), *Proceedings of the 20th international joint conference on artificial intelligence* (*IJCAI'07*) (pp. 2318–2323). Menlo Park: AAAI.
26. Järvisalo, M. (2007). *Equivalence checking multiplier designs*. SAT Competition 2007 benchmark description. http://www.tcs.hut.fi/~mjj/benchmarks/.
27. Järvisalo, M., & Junttila, T. (2007). Limitations of restricted branching in clause learning. In C. Bessiere (Ed.), *Proceedings of the 13th international conference on principles and practice of constraint programming (CP 2007). Lecture notes in computer science* (Vol. 4741, pp. 348–363). New York: Springer.
28. Järvisalo, M., & Junttila, T. (2008). On the power of top-down branching heuristics. In *Proceedings of the 23rd AAAI conference on artificial intelligence* (*AAAI-08*) (pp. 304–309). Menlo Park: AAAI.
29. Järvisalo, M., Junttila, T., & Niemelä, I. (2005). Unrestricted vs restricted cut in a tableau method for Boolean circuits. *Annals of Mathematics and Artificial Intelligence, 44*(4), 373–399.
30. Järvisalo, M., & Niemelä, I. (2008). The effect of structural branching on the efficiency of clause learning SAT solving: An experimental study. *Journal of Algorithms, 63*(1–3), 90–113.
31. Junttila, T. A., & Niemelä, I. (2000). Towards an efficient tableau method for boolean circuit satisfiability checking. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. K. Lau, C. Palamidessi, et al. (Eds.), *Proceedings of the 1st international conference on computational logic* (*CL'00*). *Lecture notes in computer science* (Vol. 1861, pp. 553–567). New York: Springer.
32. Jussila, T., Heljanko, K., & Niemelä, I. (2005). BMC via on-the-fly determinization. *International Journal on Software Tools for Technology Transfer, 7*(2), 89–101.
33. Kautz, H. A., & Selman, B. (1992). Planning as satisfiability. In B. Neumann (Ed.), *Proceedings of the 10th European conference on artificial intelligence (ECAI'92)* (pp. 359–363). New York: Wiley.
34. Krajíček, J. (1995). Bounded arithmetic, propositional logic, and complexity theory. In *Encyclopedia of mathematics and its applications* (Vol. 60). Cambridge: Cambridge University Press.
35. Kuehlmann, A., Ganai, M. K., & Paruthi, V. (2001). Circuit–based Boolean reasoning. In *Proceedings of the 38th design automation conference (DAC'01)* (pp. 232–237). New York: ACM.
36. Latvala, T., Biere, A., Heljanko, K., & Junttila, T. A. (2004). Simple bounded LTL model checking. In A. J. Hu & A. K. Martin (Eds.), *Proceedings of the 5th international conference on formal methods in computer-aided design (FMCAD'04). Lecture notes in computer science* (Vol. 3312, pp. 186–200). New York: Springer.
37. Li, C. M., & Anbulagan (1997). Heuristics based on unit propagation for satisfiability problems. In M. Pollack (Ed.), *Proceedings of the 15th international joint conference on artificial intelligence (IJCAI'97)* (pp. 366–371). San Francisco: Morgan Kaufmann.
38. Marques-Silva, J., & Guerra e Silva, L. (2003). Solving satisfiability in combinational circuits. *IEEE Design & Test of Computers 20*(4), 16–21.
39. Marques-Silva, J. P., & Sakallah, K. A. (1999). GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers, 48*(5), 506–521.
40. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th design automation conference* (*DAC'01*) (pp. 530–535). New York: ACM.

41. Nikolenko, S. I. (2005). Hard satisfiable instances for DPLL-type algorithms. *Journal of Mathematical Sciences, 126*(3), 1205–1209.
42. Papadimitriou, C. H. (1995). *Computational complexity*. Reading: Addison-Wesley.
43. Pyhälä, T. (2004). *Factoring benchmarks for SAT-solvers*. http://www.tcs.hut.fi/Software/genfacbm/.
44. Robinson, J. A. (1965). A machine oriented logic based on the resolution principle. *Journal of the ACM, 12*(1), 23–41.
45. Strichman, O. (2000). Tuning SAT checkers for bounded model checking. In E. A. Emerson & A. P. Sistla (Eds.), *Proceedings of the 12th international conference on computer aided verification (CAV'00). Lecture notes in computer science* (Vol. 1855, pp. 480–494). New York: Springer.
46. Thiffault, C., Bacchus, F., & Walsh, T. (2004). Solving non-clausal formulas with DPLL search. In M. Wallace (Ed.), *Proceedings of the 10th international conference on principles and practice of constraint programming (CP'04). Lecture notes in computer science* (Vol. 3258, pp. 663–678). New York: Springer.
47. Tseitin, G. S. (1983). On the complexity of derivation in propositional calculus. In A. Slisenko (Ed.), *Studies in constructive mathematics and mathematical logic, part II. Seminars in mathematics, V.A. Steklov mathematical institute, Leningrad* (Vol. 8, pp. 115–125). Consultants Bureau (1969). English translation appears in J. Siekmann and G. Wrightson, editors, Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970 pages 466–483. New York: Springer.
48. Urquhart, A. (1987). Hard examples for resolution. *Journal of the ACM, 34*(1), 209–219.
49. Urquhart, A. (1995). The complexity of propositional proofs. *Bulletin of Symbolic Logic, 1*(4), 425–467.
50. Velev, M. N., & Bryant, R. E. (1999). Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In L. Pierre & T. Kropf (Eds.), *Correct hardware design and verification methods, proceedings of the 10th IFIP WG 10.5 advanced research working conference (CHARME'99). Lecture notes in computer science* (Vol. 1703, pp. 37–53). New York: Springer.
51. Williams, R., Gomes, C. P., & Selman, B. (2003). Backdoors to typical case complexity. In G. Gottlob & T. Walsh (Eds.), *Proceedings of the eighteenth international joint conference on artificial intelligence (IJCAI'03)* (pp. 1173–1178). San Francisco: Morgan Kaufmann.
52. Zhang, L., Madigan, C. F., Moskewicz, M. W., & Malik, S. (2001). Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 international conference on computer-aided design (ICCAD'01)* (pp. 279–285). New York: ACM.