Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 107
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 107
Espoo 2007 HUT-TCS-A107

IMPACT OF RESTRICTED BRANCHING ON CLAUSE LEARNING SAT SOLVING

Matti Järvisalo

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 107
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 107
Espoo 2007 HUT-TCS-A107

IMPACT OF RESTRICTED BRANCHING ON CLAUSE LEARNING SAT SOLVING

Matti Järvisalo

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FI-02015 TKK, FINLAND
Tel. +358 9 451 1
Fax. +358 9 451 3369
E-mail: lab@tcs.tkk.fi
URL: http://www.tcs.tkk.fi/

ISBN 978-951-22-8907-3
ISSN 1457-7615

Multiprint Oy
Espoo 2007

ABSTRACT: Propositional satisfiability (SAT) solving procedures (or SAT solvers) are used as efficient back-end search engines in solving industrial-scale problems such as automated planning and verification. Typical SAT solvers are based on the Davis–Putnam–Logemann–Loveland (DPLL) procedure, which performs an intelligent depth-first search over the solution space of a propositional logic formula. A key technique in SAT solvers is clause learning, which has been shown to make DPLL more efficient both theoretically and in practice. Branching heuristics, that is, deciding on which variable to next set a value during search, also plays an important role in the efficiency of search.

This report investigates the effect of structure-based branching restrictions on the efficiency of modern DPLL based SAT solvers with clause learning. Branching restrictions force the solver to restrict its decision making to a subset of the problem variables. Ideally, if the branching restriction consists of structurally important variables, the solver is guided to making relevant decisions, decreasing the time needed for solving the problem instance.

The contributions of the report are two-fold. On one hand, a theoretical investigation of the effect of so called input restricted branching on the proof complexity theoretic efficiency of the proof system underlying DPLL based SAT solvers with clause learning is presented. It is shown that with input restricted branching, clause learning DPLL is polynomially incomparable with the standard DPLL. This implies that all implementations of clause learning DPLL, even with optimal heuristics, have the potential of suffering a notable efficiency decrease when input restricted branching is applied.

On the other hand, an extensive experimental evaluation of the effect of different structure-based branching restrictions on the practical efficiency of a state-of-the-art clause learning DPLL implementation is provided. Compared to experimental evaluations found in the literature, the treatment is both deeper and wider: it is not limited to comparing running times and it provides more detailed experimental evidence for the reasons why input restricted branching has the potential of reducing search efficiency. The work also investigates the effect of restricting branching in a controlled way based on structural properties other than the plain input restriction.

KEYWORDS: Boolean circuits, branching heuristics, clause learning, constraint solving, DPLL, experimentation, problem structure, proof complexity, propositional satisfiability

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Considerable advances in fundamentals and implementational techniques for constraint-based declarative problem solving have established constraint-based methods as competitive and even dominant compared to more specific algorithmic approaches for solving computationally difficult problems in a wide range of applications. Propositional satisfiability (SAT) solving procedures (or *SAT solvers*, see, e.g., [96] for a recent review, and [22] for a comparison with more general constraint satisfaction techniques, CSP), in particular, have been found to be extremely efficient as back-end search engines in solving large industrial-scale combinatorial problems. Typical examples of such real-world application domains of SAT solvers include automated planning [73, 74, 107, 72, 27], bounded model checking (BMC) of hardware and software [19, 20, 78, 12], and electronic design automation applications such as automated test pattern generation (ATPG) [80, 125, 128] and symbolic trajectory evaluation (STE) [109]. Most recently, we have witnessed applications of SAT solving techniques in new exciting fields such as bioinformatics [87, 130] and logical cryptanalysis [92, 94, 37] and model checking of security protocols [11].

When solving problems with tools provided by propositional satisfiability, the problem at hand is typically encoded as a *conjunction of disjunctions of Boolean variables and their negations*, or, in other words, as a propositional logic formula in *conjunctive normal norm* (CNF, or *clausal* form) [123]. This is due to the fact that typically SAT solvers take CNF as input (although non-clausal solvers have also been suggested and implemented, see e.g. [69, 79, 43, 64, 88, 129]). However, due to difficulty of *modeling* problems as CNF formulas, direct CNF encodings are rarely used. The problem at hand is typically encoded as a general propositional formula ϕ, which is then translated into a logically equivalent CNF formula by introducing additional variables for the subformulas of ϕ. This is often referred to as the "Tseitin" translation [131].

SAT solving procedures can be divided into *stochastic local search based* and *complete* (or *systematic*) solvers. Local seach procedures (see, e.g., [121, 120, 119, 93, 59, 58, 118, 56, 24]) are based on iterating over a current solution candidate by *flipping* value assignments in the candidate, typically based on a *neighborhood* function. Such procedures are characterized by their inability to show (generate a *proof* for) the non-existence of solutions, although recently also local search for *unsatisfiability* has also been considered [105]. While local search SAT solvers have proven very successful in solving *random* satisfiability problem instances (see, e.g., [117]), the breakthroughs in applying SAT solvers in relevant *structural* real-world problem domains are due to *complete SAT solving procedures*, on which we will also concentrate in this report.

Although some investigations into applying local search solvers in structural problem domains have been made [114, 71, 124, 103], the most successful SAT solvers aimed at solving structured problems are based on the complete *Davis–Putnam–Logemann–Loveland* procedure (DPLL) [36, 35], which works on CNF input. Such solvers perform an *intelligent* search over the whole assignment space (or *search space*), and can provide *unsatisfiability proofs* in case there are no solutions to the problem instance at hand. The relevance of DPLL solvers is nowadays further highlighted by their application as the core solver engine in the *Satisfiability Modulo Theories approach* (SMT) [115, 98, 23], where the input language is enriched with more expressive constraint types, such as *linear equations*, to allow Boolean combinations of such constraints.

As SAT solvers have become a standard tool for solving various increasingly difficult industrial problems, there is a demand for more and more robust and efficient solvers. Research on boosting the efficiency of DPLL solvers has concentrated on incorporating techniques such as *intelligent branching heuristics* (e.g., [57, 84, 97]), novel *propagation mechanisms* (e.g., *binary clause reasoning* [14] and *equivalence reasoning* [83, 55]), efficient propagator implementations (*watched literals* [97]), *randomization* and *restarts* [68, 49], and *clause learning* [90] into DPLL. Out of these concepts, clause learning can be regarded as the most important progressive step, as witnessed by a sequence of further improved solvers [68, 90, 97, 48, 41], and also by proof complexity theoretic efficiency analysis [18]. While new propagation mechanisms, such as equivalence reasoning, have been successfully implemented into DPLL, most clause learning solvers still rely on standard *unit propagation* as the sole propagator. The integration of more sophisticated propagators with clause learning is not trivial, and typically DPLL based solvers with equivalence reasoning do not incorporate clause learning. As for intelligent decision (or *branching*) heuristics, while solvers without clause learning incorporate heuristics based on literal counting [57] and/or one-step lookahead [84, 54, 9], branching in clause learning solvers is also driven by learning. Most clause learning solvers implement variations of—or build on top of, see e.g. [41, 48, 112]—the *variable state independent decaying sum* (VSIDS) heuristic [97], which values the variables that have played an active role in reaching recent conflicts. Moreover, clause learning enables *non-chronological backtracking* (or *backjumping*). In fact, as noted, e.g., in [60], since search space traversal is guided tightly by clause learning in modern solvers with the help of unit propagation and restarts, clause learning solvers can be seen as performing a process quite unlike the search performed by implementations of the basic DPLL.

Nevertheless, branching heuristics, i.e., deciding on which variable to next set a value during search, plays an important role in the efficiency of search. Due to an increasing need for solving large structural problems, techniques for making efficiency-improving decisions are vital. Intuitively, the inherent structure of the problem domain is reflected in the importance of individual variables. Irrelevant decisions may have an exponential effect on the running times of SAT solvers.

In addition to developing more effective (*dynamic*) branching heuristics, another complementary view on branching is provided by the concept of *(static) branching restrictions*. In SAT based approaches to structured problems such as bounded model checking (of both hardware and software) and automated planning, the CNF encoding is often derived from a transition relation, where the behaviour of the underlying system is dependent on the *input*—initial state, nondeterministic choices, et cetera—of the system. Experimental case studies in specific problem domains [46, 126, 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 variables* (sometimes referred to as *independent variables*), corresponding to the input of the underlying system, by letting the solver then apply its own dynamic heuristics to this set of variables. We will refer to this restriction as the *input-restriction*, or *input-restricted branching*. Since the system behaviour is determined by its input, input-restricted branching DPLL remains complete. Intuitively, this changes the worst-case behaviour of DPLL from the order of 2^N to 2^I with $I << N$, where I and N are the number of input variables and all variables in the CNF encoding, respectively.

From another point of view, one can investigate the *best-case* performance of SAT algorithms through *proof complexity* [32], 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 propositional proof systems S,S', we say that S' *(polynomially) simulates* S if, for all infinite families $\{F_n\}$ of unsatisfiable formulas, there is a polynomial that bounds for all F_n the length of the shortest proofs in S' with respect to 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.

Any implementation of the basic DPLL procedure can be seen as a determinization of a DPLL *proof system*. Recently clause learning DPLL has also been characterized as a proof system called CL [18]. Through this characterization, Beame et al. [18] show that CL can provide exponentially 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 [67]: input-restricted branching DPLL cannot simulate DPLL. The question for DPLL with clause learning is left open.

Considering the experimental side of branching restrictions, the case studies on restricted branching that we are aware of, including [46, 126, 45, 122], consider mainly input-restricted branching as the only structural way of restricting the decision making in SAT solvers, and concentrate usually only on running times of solvers. The existing literature sheds little light on the effect of the restriction to the inner workings of SAT solvers, and, in many cases, current state-of-the-art solver techniques are not used. This is important to notice due to the fundamental difference between non-clause learning and clause learning solvers.

1.1 CONTRIBUTIONS

In this report we study the effect of structurally restricting branching on the efficiency of clause learning DPLL SAT solving from a theoretical as well as practical point-of-view. As detailed below, the main contributions are:

- We extend the result in [67], stating that input-restricted branching DPLL cannot simulate DPLL, to the case of clause learning DPLL: we show that input-restricted branching CL and the basic DPLL without clause learning are polynomially incomparable. Hence, input-restricted branching CL cannot simulate CL.

- We provide an experimental evaluation of the effect of structure-based branching restrictions on the efficiency of a state-of-the-art clause learning SAT solver.

1.1.1 Relative Efficiency of Branching Restricted Clause Learning

We investigate the proof complexity theoretic efficiency of input-restricted branching 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 simulate the basic DPLL (whichdoes not apply clause learning). This is surprising, since the unrestricted version of this variant of CL can efficiently simulate general resolution [18], being thus very powerful compared to DPLL.

1.1.2 Experimental Evaluation of Restricted Branching

We present an extensive experimental evaluation of the effect of structure-based branching restrictions on the efficiency of solving structural SAT instances. The emphasis is on the interplay between structure-based branching restrictions and typical clause learning based search techniques in modern complete SAT solvers:

(i) We perform an in-depth investigation into the effect of input-restricted branching on the effectiveness of clause learning SAT solver techniques, including the often applied VSIDS heuristic and the effectiveness of learned conflict clauses.

(ii) In order to study whether the robustness of input-restricted branching can be improved, we devise and apply controlled schemes for allowing branching additionally on CNF variables other than inputs based on structural properties of non-clausal formulas, such as the number of occurrences of sub-formulas.

The results show that by restricting the set of branchable variables to input variables, the effectiveness of the clause learning bound VSIDS heuristic and conflict clauses weakens. However, by selectively allowing branching on additional variables based on structural properties, branching can be restricted rather heavily without losing the efficiency of the original unrestricted branching solver. However, it seems unlikely that branching restrictions can make modern clause learning solvers more efficient in general without coupling them more tightly with clause learning techniques.

This study complements known experimental studies on comparing SAT solver techniques, such as clause learning schemes [137], restarts [60], and comparisons of branching heuristics (e.g., [57, 89]). Our aim is to provide a more coherent picture of the effect of branching restrictions on the inner workings of modern clause learning solvers, and to understand how important underlying structural properties of variables are in making decisions in clause learning SAT solvers.

1.2 RELATED WORK

1.2.1 Experiments on Branching Restrictions

In the context of SAT based scheduling, the possibility of restricting branching to inputs (or *control variables*) is suggested in [33], without empirical evaluation, however. For SAT based planning, input-restricted branching (or branching on *action variables*) is considered in [46], showing that the DPLL solver Tableau (having *no clause learning*) benefits from this restriction on the considered instances. Considering SAT based bounded model checking (BMC), in [126] input-restricted branching (or branching on *model variables*) is applied with the clause learning solver Grasp, in which the decision heuristic is *not coupled with* clause learning. Additionally, the work concentrates on comparing the efficiency of SAT and BDD based BMC. In [45], the authors investigate the effect of restricting branching to inputs (or *independent variables*, calling this the *independent variables set (IVS) heuristic*) on planning, BMC, and crafted SAT instances using the SAT solver Sim. The presented results deal partly with clause learning. However, the emphasis of the work in [45] is on comparing different decision heuristics that are *not coupled with* clause learning, as opposed to the popular VSIDS heuristic today. Most recently, in the context of SAT based automated test pattern generation (ATPG), in [122] the authors investigate the effect of input-restricted branching on the efficiency of a variety of modern clause learning solvers. In addition, the authors also consider *fanout-restricted branching*, in which branching is allowed additionally on variables which are associated with subformulas occurring multiple times in the original non-clausal problem.

However, the case studies on restricted branching that we are aware of, including [46, 126, 45, 122], consider mainly input-restricted branching as the only structural way of restricting the decision making in SAT solvers. Moreover, the evaluations are based only on the running times of the solver; a more in-depth investigation into the real cause of the differences in running times with respect to the applied solver techniques is somewhat lacking. Additionally, in many cases, current state-of-the-art solver techniques are not used. This is important to notice due to the fundamental difference between non-clause learning and clause learning solvers.

Additionally, branching variable orderings for DPLL based on structural information have also been studied [61, 8]. In these works, the solver is forced to follow an order derived from structural properties of the formula, as opposed to the branching restrictions studied in this work where the solver is allowed to apply its own dynamic heuristic for branching on variables in the restriction.

Finally, [50] develops techniques for automatic extraction of functional dependencies from CNF formulas, in order to exploit input-restricted branching. The extraction of functional dependencies is additionally investigated in [110, 42].

1.2.2 Related Theoretical Results

There are also theoretical results on the effect of restricted branching on the efficiency of the underlying inference system of DPLL. In [45] it is noted that restricting to independent variables can result in exponential loss of efficiency for DPLL without clause learning. Using the notion of proof complexity, again considering DPLL without clause learning, [67] studies the effect of input-restriction and, additionally, a variety of other static and dynamic restrictions, such as *top-down branching*, which is closely related to the *justification frontier heuristic* (see e.g. [88]) used often in DPLL style Boolean circuit satisfiability solvers applied in electronic design automation (EDA). The result is a relative efficiency hierarchy for the considered restrictions, showing that, for example, input-restricted branching DPLL cannot simulate top-down branching DPLL, which in turn cannot simulate the standard (unrestricted branching) DPLL.

The complexity of making the optimal branching decision during search in DPLL is studied in [85], with the results that, while the problem for the standard DPLL is not on the first level of the polynomial hierarchy[1] (it is $\Delta_2^{\mathbf{P}}[\log n]$–hard), it may be even harder for branching restricted DPLL (that is, $\mathbf{NP}^{\mathbf{PP}}$–hard, i.e., spanning the whole polynomial hierarchy, under a certain assumption, see [85]). Moreover, the problem of determining the size of the optimal proofs in DPLL is **coNP**–hard [85].

Finally, the work in [95, 62] gives a proof complexity theoretic study of typical branching schemes (so called *2-way* and *d-way branching*) in the context of CSP solving.

1.2.3 Backdoor Sets

The concept of a *(strong) backdoor set* [136, 111] of variables is closely related to restricting branching so that the resulting solving method is still complete. A *unit propagation backdoor set for* DPLL is a set of variables such that, once all of these variables have values, all the other variables are set values by unit propagation. Thus one intuitive backdoor set is the set of input variables.

While deciding whether a backdoor set of a given size exists is intractable in general, algorithms for finding small backdoor sets for CNF formulas have been developed in [75, 63], for example.

The *parameterized complexity* [39] of finding minimal backdoor sets has also received attention. In [127], the parameterized complexity of determining whether there is a (strong and weak) backdoor set of a given size is shown to be $\mathbf{W}[\mathbf{P}]$-complete for unit propagation and *pure literal elimination*. Additionally, [100] studies the parameterized complexity of detecting backdoor

[1]For background in complexity classes and the polynomial hierarchy, see, e.g., [102].

sets with respect to the polynomially solvable SAT instance classes HORN and 2-SAT.

1.3 OUTLINE OF THE REPORT

As preliminaries, in Chapter 2 we define propositional satisfiability, and constrained Boolean circuits, which we use for representing structural formulas, and discuss the close relationship between circuits and CNF formulas. We then review the Resolution proof system and characterisations of DPLL and CL, and discuss known results concerning their relative efficiency (Chapter 3). The main theoretical and experimental results of the report are presented in Chapters 4 and 5, respectively.

2 PROPOSITIONAL SATISFIABILITY AND CONSTRAINED BOOLEAN CIRCUITS

In this chapter we review basic concepts related to propositional satisfiability and define constrained Boolean circuits which we use as the representation form for structural 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 $\neg x$ (or by $\bar{x}$ when convenient), where $\neg$ is the negation (not). As usual, we identify $\bar{\bar{x}}$ 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. 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 slight abuse of notation, if $\tau(x) = v$, then $\tau(\bar{x}) = \neg v$, 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* F if it satisfies every clause in it. 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 a logically equivalent CNF formula by introducing additional variables for the subformulas of ϕ. *Boolean circuits* (see e.g. [102]) offer a natural way of presenting propositional formulas in a compact DAG-like structure with *subformula 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 $\mathcal{C}$ 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 $\mathcal{C}$, and (ii) the underlying directed graph

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

is acyclic. If $\langle g', g\rangle \in E(\mathcal{C})$, then g' is a *child* of g and g is a *parent* of g'. If $g := f(g_1, \ldots, g_n)$ is in $\mathcal{C}$, 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 $\mathcal{C}$ is a (partial) function $\tau : G \to \{\mathbf{f}, \mathbf{t}\}$. An assignment τ is consistent with $\mathcal{C}$ if $\tau(g) = f(\tau(g_1), \ldots, \tau(g_n))$ for each $g := f(g_1, \ldots, g_n)$ in $\mathcal{C}$.

A *constrained Boolean circuit* $\mathcal{C}^\tau$ is a pair $\langle \mathcal{C}, \tau\rangle$, where $\mathcal{C}$ is a Boolean circuit and τ is a partial assignment for $\mathcal{C}$. With respect to a $\langle \mathcal{C}, \tau\rangle$, 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* $\mathcal{C}^\tau$ if (i) τ' is consistent with $\mathcal{C}$, and (ii) $\tau' \supseteq \tau$. If some assignment satisfies $\mathcal{C}^\tau$ then $\mathcal{C}^\tau$ is *satisfiable* and otherwise *unsatisfiable*.

Typical Boolean functions for gate types in Boolean circuits are:

- $\textsc{not}(g)$ evaluates to $\mathbf{t}$ if and only if g evaluates to $\mathbf{f}$.

- $\textsc{or}(g_1, \ldots, g_n)$ evaluates to $\mathbf{t}$ if and only if at least one of $g_1, \ldots, g_n$ evaluates to $\mathbf{t}$.

- $\textsc{and}(g_1, \ldots, g_n)$ evaluates to $\mathbf{t}$ if and only if all $g_1, \ldots, g_n$ evaluate to $\mathbf{t}$.

- $\textsc{imply}(g_1, g_2)$ evaluates to $\mathbf{t}$ if and only if (i) g_1 evaluates to $\mathbf{f}$, or (ii) g_2 evaluates to $\mathbf{t}$.

- $\textsc{equiv}(g_1, g_2)$ evaluates to $\mathbf{t}$ if and only if (i) both g_1, g_2 evaluate to $\mathbf{f}$, or (ii) both g_1, g_2 evaluate to $\mathbf{t}$.

- $\textsc{ite}(g_1, g_2, g_3)$ evaluates to $\mathbf{t}$ if and only if (i) g_1 and g_2 evaluate to $\mathbf{t}$, or (ii) g_1 evaluates to $\mathbf{f}$ and g_3 evaluates to $\mathbf{t}$.

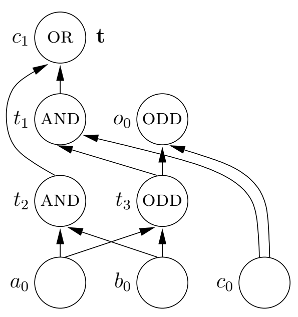
- $\textsc{even}(g_1, g_2)$ evaluates to $\mathbf{t}$ if and only if (i) both g_1, g_2 evaluate to $\mathbf{t}$, or (ii) neither of g_1, g_2 evaluates to $\mathbf{t}$.

- $\textsc{odd}(g_1, g_2)$ evaluates to $\mathbf{t}$ if and only if exactly one of g_1, g_2 evaluates to $\mathbf{t}$.

When convenient, we will identify a constrained circuit $\mathcal{C}^\tau = \langle \mathcal{C}, \tau\rangle$ with its underlying directed graph $\langle G, E(\mathcal{C})\rangle$, where the non-input gates are labeled with the corresponding gate types, and gates constrained by τ additionally with the corresponding constraint. With this intuition, we may write $\langle\langle G, E(\mathcal{C})\rangle, \tau\rangle$ instead of $\langle \mathcal{C}, \tau\rangle$.

Example 2.1 *Figure 2.1 shows a Boolean circuit for a full-adder with the constraint that the carry-out bit c_1 is $\mathbf{t}$. One satisfying truth assignment for the circuit is*

$$\{\langle c_1, \mathbf{t}\rangle, \langle t_1, \mathbf{t}\rangle, \langle o_0, \mathbf{f}\rangle, \langle t_2, \mathbf{f}\rangle, \langle t_3, \mathbf{t}\rangle, \langle a_0, \mathbf{t}\rangle, \langle b_0, \mathbf{f}\rangle, \langle c_0, \mathbf{t}\rangle\}.$$



$$\begin{aligned} \mathcal{C} \;=\; \{ & c_1 := \mathrm{OR}(t_1, t_2) \\ & t_1 := \mathrm{AND}(t_3, c_0) \\ & o_0 := \mathrm{ODD}(t_3, c_0) \\ & t_2 := \mathrm{AND}(a_0, b_0) \\ & t_3 := \mathrm{ODD}(a_0, b_0)\} \end{aligned}$$

$$\tau = \{\langle c_1, \mathbf{t}\rangle\}$$

Figure 2.1: A constrained Boolean circuit $\langle \mathcal{C}, \tau\rangle$.

For notational convenience, when well-defined, the *join* of constrained circuits $\mathcal{A}^\tau = \langle \mathcal{A}, \tau\rangle$ and $\mathcal{B}^\theta = \langle \mathcal{B}, \theta\rangle$ is $\mathcal{A}^\tau \cup \mathcal{B}^\theta := \langle \mathcal{A} \cup \mathcal{B}, \tau \cup \theta\rangle$. 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. Additionally, for a constrained circuit $\langle\langle G, E\rangle, \tau\rangle$, the *sub-circuit* $\langle\langle G', E'\rangle, \tau'\rangle$ induced by a $G' \subseteq G$ is defined by $E' = \{\langle g, g'\rangle \in E \cap (G' \times G')\}$ and $\tau' = \{\langle g, \epsilon\rangle \in \tau \mid g \in G'\}$.

2.2.1 Note on Representation Forms for Structural Formulas

Many graph-based representation forms for propositional formulas can be found in the literature. Some of these, [79, 1, 135] for example, can be seen as special cases of our definition for Boolean circuits. *AND-inverter graphs* (AIGs) [79] are basically Boolean circuits in which only the gate types AND and NOT are used. It should be noted that typically work on such graph presentation forms (e.g., [44, 79, 1, 10, 21]) deals with techniques for reducing the size of circuits in order to enable storing very large formulas. This is also the case with *reduced Boolean circuits* (RBCs), as defined in [1]. Recently, Boolean circuits have also been called *propositional DAGs* [135], although the underlying formalism coincides with Boolean circuits. However, in this report we concentrate on investigating the relative efficiency of solver techniques for CNF formulas, and apply graph-based representation of propositional formulas only for representing structure in the formulas. Hence we use our rather standard definition of Boolean circuits.

Another often applied representation form for propositional formulas is offered by *binary decision diagrams* (BDDs) [82, 4]. As opposed to Boolean circuits, BDDs explicitly represent the *decision tree* (or *truth table*) of a propositional formula as a DAG using only the *if-then-else* structure (ITE function). BDDs are build recursively by applying the so called *Shannon expansion*, which does a case analysis by expanding on a selected variable. In *ordered BDDs* (OBDDs, for a detailed discussion, see [26, 40]), the expansion is done using a fixed ordering on the variables. *Reduced OBDDs* (ROBDDs) provide a *canonical* OBDD presentation of any propositional formula, i.e., there is exactly one ROBDD for any formula. After building the ROBDD of a propositional formula, the satisfiability of the formula can be checked in *constant time*. Generally, however, the biggest disadvantage of BDDs is in the building process of an ROBDD; most notably, the space consumption in the process can grow exponentially in the size of the propositional formula. In fact, the algorithms for building (reduced) OBDDs can be seen as an inference system, which is rather different to the DPLL based methods on which we concentrate in this report. For studies on OBDDs as a proof system from the view point of proof complexity, see [51, 13, 77, 116].

As a final remark, *Boolean expression diagrams* (BEDs) [10] aim at extending BDDs with Boolean circuit style gate types. For a short overview of BDDs, BEDs, RBCs, and AIGs, we refer the reader to [21]. Since the DPLL based reasoning methods of interest here deal with CNF formulas, we will next define the translation from Boolean circuits to CNF formulas.

2.3 TRANSLATING BOOLEAN CIRCUITS TO CNF

In order to exploit clausal SAT solvers in solving instances of Boolean circuit satisfiability, the circuit has to be translated to CNF.

In this report we apply the following variation of the standard "Tseitin-style" [131] translation. 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 2.1. Given a constrained Boolean circuit $\mathcal{C}^\tau$, we will denote its CNF translation by $\mathsf{cnf}(\mathcal{C}^\tau)$.

Table 2.1: CNF translation for constrained Boolean circuits

gate g	clauses for $g \Rightarrow f(g_1, \ldots g_n)$	clauses for $f(g_1, \ldots g_n) \Rightarrow g$
$g := \text{IMPLY}(g_1, g_2)$	$(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2)$	$(\tilde{g} \vee \tilde{g}_1), (\tilde{g} \vee \neg\tilde{g}_2)$
$g := \text{EQUIV}(g_1, g_2)$	$(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2), (\neg\tilde{g} \vee \tilde{g}_1 \vee \neg\tilde{g}_2)$	$(\tilde{g} \vee \neg\tilde{g}_1 \vee \neg\tilde{g}_2), (\tilde{g} \vee \tilde{g}_1 \vee \tilde{g}_2)$
$g := \text{EVEN}(g_1, g_2)$	$(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2), (\neg\tilde{g} \vee \tilde{g}_1 \vee \neg\tilde{g}_2)$	$(\tilde{g} \vee \neg\tilde{g}_1 \vee \neg\tilde{g}_2), (\tilde{g} \vee \tilde{g}_1 \vee \tilde{g}_2)$
$g := \text{ODD}(g_1, g_2)$	$(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \neg\tilde{g}_2), (\neg\tilde{g} \vee \tilde{g}_1 \vee \tilde{g}_2)$	$(\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2), (\tilde{g} \vee \tilde{g}_1 \vee \neg\tilde{g}_2)$
$g := \text{OR}(g_1, \ldots, g_n)$	$(\neg\tilde{g} \vee \tilde{g}_1 \vee \cdots \vee \tilde{g}_n)$	$(\tilde{g} \vee \neg\tilde{g}_1),\ldots,(\tilde{g} \vee \neg\tilde{g}_n)$
$g := \text{AND}(g_1, \ldots, g_n)$	$(\neg\tilde{g} \vee \tilde{g}_1),\ldots,(\neg\tilde{g} \vee \tilde{g}_n)$	$(\tilde{g} \vee \neg\tilde{g}_1 \vee \cdots \vee \neg\tilde{g}_n)$
$g := \text{NOT}(g_1)$	$(\neg\tilde{g} \vee \neg\tilde{g}_1)$	$(\tilde{g} \vee \tilde{g}_1)$
$g := \text{ITE}(g_1, g_2, g_3)$	$(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2), (\neg\tilde{g} \vee \tilde{g}_1 \vee \tilde{g}_3)$	$(\tilde{g} \vee \neg\tilde{g}_1 \vee \neg\tilde{g}_2), (\tilde{g} \vee \tilde{g}_1 \vee \neg\tilde{g}_3)$
$\langle g, \mathbf{t} \rangle \in \tau$	$(\tilde{g})$	
$\langle g, \mathbf{f} \rangle \in \tau$	$(\neg\tilde{g})$	

2.3.1 Note on Circuit-to-CNF Encodings

The "standard translation" from constrained Boolean circuits to CNF formulas presented in Table 2.1 is often (depending on context) referred to as "Tseitin translation", as it follows the encoding of arbitrary propositional formulas as CNF formulas presented in [131]. Similarly, it could also be regarded as "Cook translation", as the same idea is used in [30] in proving the **NP**–completeness of SAT. A well–known refinement of this standard encoding is the Plaisted–Greenbaum *polarity exploiting translation* [104]. Compact CNF encodings of Boolean circuits (or non-clausal formulas) are also developed in [38, 101, 65], for example. However, within the scope of this report, we will apply the standard translation, as our main interests lie in studying the effect of restricted branching, rather than comparing different translations.

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, $\mathsf{circuit}(F) := \langle \mathcal{C}, \tau \rangle$ is defined by associating an input gate g_x with each $x \in \mathsf{vars}(F)$, a NOT-gate $g_{\bar{x}}$ 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

$$\begin{aligned}\mathcal{C} \;:=\; & \{g_F := \mathrm{AND}(g_{C_1}, \ldots, g_{C_k})\} \cup \{g_{\bar{x}} := \mathrm{NOT}(g_x) \mid x \in \mathsf{vars}^-(F)\} \cup \\ & \{g_{C_i} := \mathrm{OR}(g_{l_{i,1}}, \ldots, g_{l_{i,n_i}}) \mid C_i = \{l_{i,1}, \ldots, l_{i,n_i}\} \in F\}.\end{aligned}$$

Example 2.2 $\mathsf{circuit}(\{\{a, b\}, \{a, \bar{b}\}, \{\bar{a}, b\}, \{\bar{a}, \bar{b}\}\})$ *is shown in Figure 2.2.*

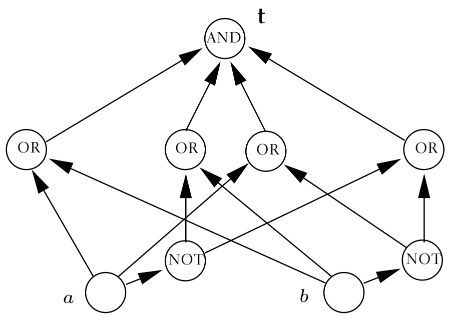


Figure 2.2: The Boolean circuit $\mathsf{circuit}(\{\{a, b\}, \{a, \bar{b}\}, \{\bar{a}, b\}, \{\bar{a}, \bar{b}\}\})$

3 CNF PROOF SYSTEMS AND SAT SOLVING

In this chapter 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 needed 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 [36, 35] 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 report.

3.1 PROPOSITIONAL PROOF SYSTEMS AND COMPLEXITY

Formally, a *propositional proof system* [32] 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 $S(F, p)$. 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 a too weak proof system. As a measure of hardness of proving unsatisfiability of a CNF formula F in a proof system S, the *(proof) complexity* 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 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,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, where C_S and $C_{S'}$ are the complexities of F_n in S and S', respectively. 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 minimal length proofs in S for $\{F_n\}$ are asymptotically superpolynomially longer that the minimal 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 [99, 3, 2, 6], a satisfying truth assignment always acts as a *polynomial length* witness for the satisfiability of an arbitrary satisfiable formula F.

3.2 RESOLUTION

The well-known Resolution proof system [108] (RES) is based on the *resolution rule*. Let C, D be clauses, and x a Boolean variable. The resolution rule is

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

or, in other words, we can *directly derive* $C \cup D$ from $\{x\} \cup C$ and $\{\bar{x}\} \cup D$ 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 $j, k < i$ (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, inner nodes are derived clauses, and the root is the clause C_m. 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 subsequently used multiple times in the proof, must be derived anew each time starting from initial clauses. Other well-known refinements include *regular resolution* [131] (any variable can be resolved upon at most once along any path in the proof from an initial clause to $\emptyset$), *Davis–Putnam* (or *ordered*) *resolution* [36] (a refinement of regular resolution where every sequence of variables resolved on in a path from an initial clause to $\emptyset$ respects the same ordering on the variables), and *linear resolution* [28] (each clause C_i must be either an initial clause or be directly derived from C_{i-1} and C_j, where $j < i - 1$).

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 [29, 53, 131, 132, 34, 5, 15, 16] 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 less than m holes so that every pigeon has its own hole). We will consider the case $m = n + 1$ encoded as the CNF formula

$$\mathrm{PHP}_n^{n+1} := \bigwedge_{i=1}^{n+1} \Big(\bigvee_{j=1}^{n} p_{i,j} \Big) \wedge \bigwedge_{j=1}^{n} \bigwedge_{i=1}^{n} \bigwedge_{i'=i+1}^{n+1} (\bar{p}_{i,j} \vee \bar{p}_{i',j}),$$

where each $p_{i,j}$ is a Boolean variable with the interpretation "$p_{i,j}$ is **t** if and only if the i^{th} pigeon sits in the j^{th} hole".

Theorem 3.1 (Haken [53]) *There is no polynomial length* RES *proof of the formula* PHP_n^{n+1}.

It is also known that T-RES is a *proper* refinement of RES. This originates from the facts that *regular resolution* cannot simulate RES [47, 7], and T-RES in turn cannot simulate regular resolution [133].

Corollary 3.1 ([47, 133]) T-RES *cannot polynomially simulate* RES.

3.3 THE DAVIS–PUTNAM–LOGEMANN–LOVELAND PROCEDURE

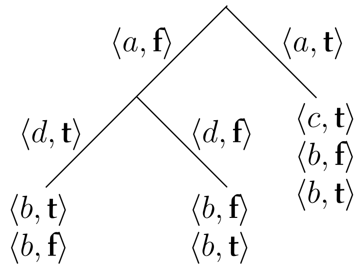
Most modern complete SAT solvers are based on the Davis–Putnam–Logemann–Loveland (or DPLL) procedure [36, 35]. Given a CNF formula F as input, DPLL is a depth-first search procedure building a partial assignment τ for the variables in F through (i) *branching* and (ii) *unit propagation* (UP). In branching, the current assignment τ is extended with the assignment (*decision*) $\langle x, v\rangle$, where v is either **f** of **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 **f** and **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.

Example 3.1 *Consider the CNF formula*

$$\phi := \{\{\bar{a}, c\},\ \{\bar{a}, \bar{b}, \bar{c}\},\ \{\bar{a}, b, \bar{c}\},\ \{a, b, d\},\ \{a, b, \bar{d}\},\ \{a, \bar{b}, d\},\ \{a, \bar{b}, \bar{d}\}\}.$$

A DPLL *search for this unsatisfiable formula is presented in Figure 3.1. Assuming that* DPLL *has traversed the tree in prefix order, the assignment* $\langle a, \mathbf{f}\rangle$ *and* $\langle d, \mathbf{t}\rangle$ *have led to a conflict in variable* b *by unit propagation. Thus* DPLL *backtracks by assigning* $\langle d, \mathbf{f}\rangle$*, and after the resulting conflict by assigning* $\langle a, \mathbf{t}\rangle$.



$\langle a, \mathbf{f}\rangle$
$\langle a, \mathbf{t}\rangle$
$\langle d, \mathbf{t}\rangle$
$\langle d, \mathbf{f}\rangle$
$\langle c, \mathbf{t}\rangle$
$\langle b, \mathbf{f}\rangle$
$\langle b, \mathbf{t}\rangle$
$\langle b, \mathbf{t}\rangle$
$\langle b, \mathbf{f}\rangle$
$\langle b, \mathbf{f}\rangle$
$\langle b, \mathbf{t}\rangle$

Figure 3.1: A DPLL search tree for the CNF formula ϕ

3.3.1 DPLL and T-RES

Consider an arbitrary CNF formula F. From the proof theoretic point of view, DPLL can be seen as a tableau proof system [91] with two rules: the *branching rule*

$$\frac{x \in \mathsf{vars}(F)}{x \quad \mid \quad \bar{x}}$$

and the unit clause rule

$$\frac{\begin{array}{c}(l \vee l_1 \vee l_2 \vee \cdots \vee {}_k) \in F\\ \bar{l}_1\\ \vdots\\ \bar{l}_k\end{array}}{l}$$

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 $\bar{x}$. 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 $\bar{x}$ 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.

Example 3.2 *Recall the CNF formula ϕ in Example 3.1. The* DPLL *search shown in Figure 3.1 seen as a tableau proof is shown in Figure 3.2.*

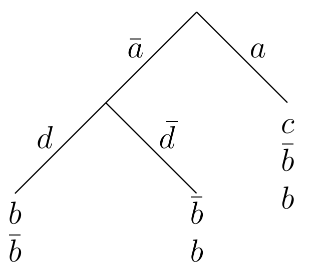


Figure 3.2: A DPLL (tableau) proof of the CNF formula ϕ in Example 3.1

One-step lookahead (see, e.g., [84]) 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 $\bar{v}$. This technique does not add to the strength of DPLL, since the same effect can obviously be accomplished by branching on x. Notice that one-step lookahead has also been used for updating the decision heuristic values of variables [84, 55], for example, so that heuristic values as based on the number of new assignments deduced by unit propagation when adding $\langle x, v\rangle$ to τ.

It is well-known that DPLL and T-RES can polynomially simulate each other (see [17] for example). One can show that for any unsatisfiable CNF formula, a DPLL proof, with applications of the unit clause rule "simulated by branching", always corresponds one-to-one with a T-RES proof, and vice versa.

Fact 3.1 DPLL *and* T-RES *are polynomially equivalent.*

In more detail, any DPLL proof can be seen as a T-RES proof with the intuition that the strength of DPLL does not really depend on whether the unit clause rule is applied. Any application of the unit clause rule on a clause C can be seen as branching on the remaining unassigned literal $l \in C$ in the sense that, by assigning **f** to l by branching, C is falsified by the current assignment.

Example 3.3 *Recall the CNF formula ϕ in Example 3.1. The* DPLL *proof shown in Figure 3.2, with applications of the unit clause rule seen as branching, is shown in Figure 3.3. The leafs of the tree are labeled with the respective unsatisfiable clauses in ϕ under each branch.*

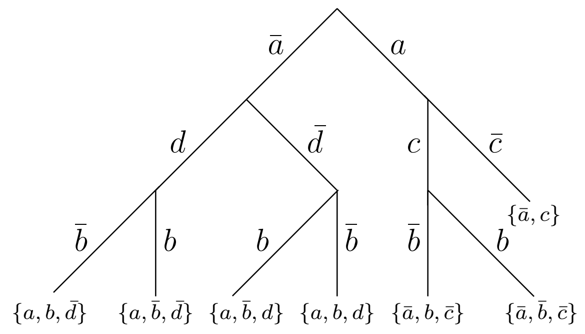


Figure 3.3: The DPLL proof in Figure 3.2 with applications of the unit clause rule seen as branching

The next example shows how such DPLL proofs can be seen as T-RES proofs.

Example 3.4 *Figure 3.4 shows the* T-RES *proof corresponding to the* DPLL *proof in Figure 3.3, where at each step the resolution rule is applied by resolving on the branching variable.*

3.3.2 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.

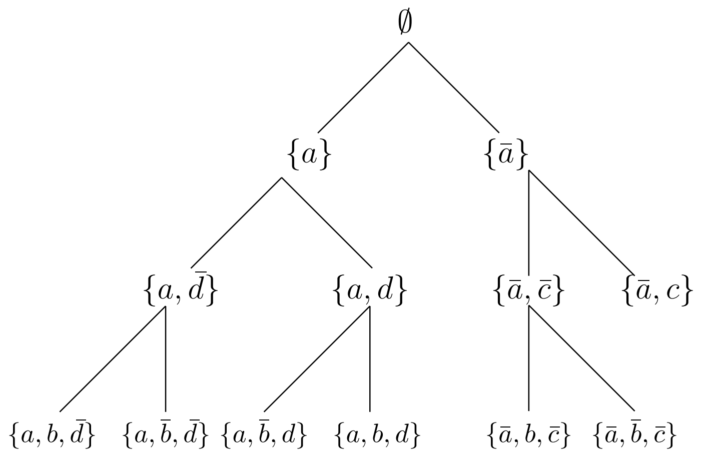


Figure 3.4: The T-RES proof corresponding to the DPLL proof in Figure 3.3

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.

Example 3.5 *Recall the* DPLL *proof in Figure 3.2. In the branch with* $(a, c, \bar{b}, b)$ *all literals are on decision level 1, the variable* a *is the sole decision variable in the branch, and* $c, \bar{b}, b$ *are implied literals. In the branch with* $(\bar{a}, d, b, \bar{b})$*, the literal* $\bar{a}$ *is on decision level 1 and the other literals are on decision level 2.*

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 deducing additional literals to L with the unit clause rule alone.

Definition 3.1 *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*

$$\begin{aligned} E &= \{\langle l_i, l \rangle \mid \{\bar{l}_1, \ldots, \bar{l}_k, l\} \in F \text{ and } l_1, \ldots, l_k \in V\} \cup \\ &\quad \{\langle x, \Lambda \rangle, \langle \bar{x}, \Lambda \rangle \mid x, \bar{x} \in V\}. \end{aligned}$$

For a given implication graph, any variable x with $x, \bar{x} \in V$ is called a *conflict variable*, and $x, \bar{x}$ 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 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 necessary forced systematically when backtracking. In other words, branching in CL 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 $\bar{x}$.

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 3.2 *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 $x, \bar{x}$.*

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 $\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_k$ 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 on the choice of the conflict graph.

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 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 3.6 *A hypothetical conflict graph is illustrated in Figure 3.5. 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 three*

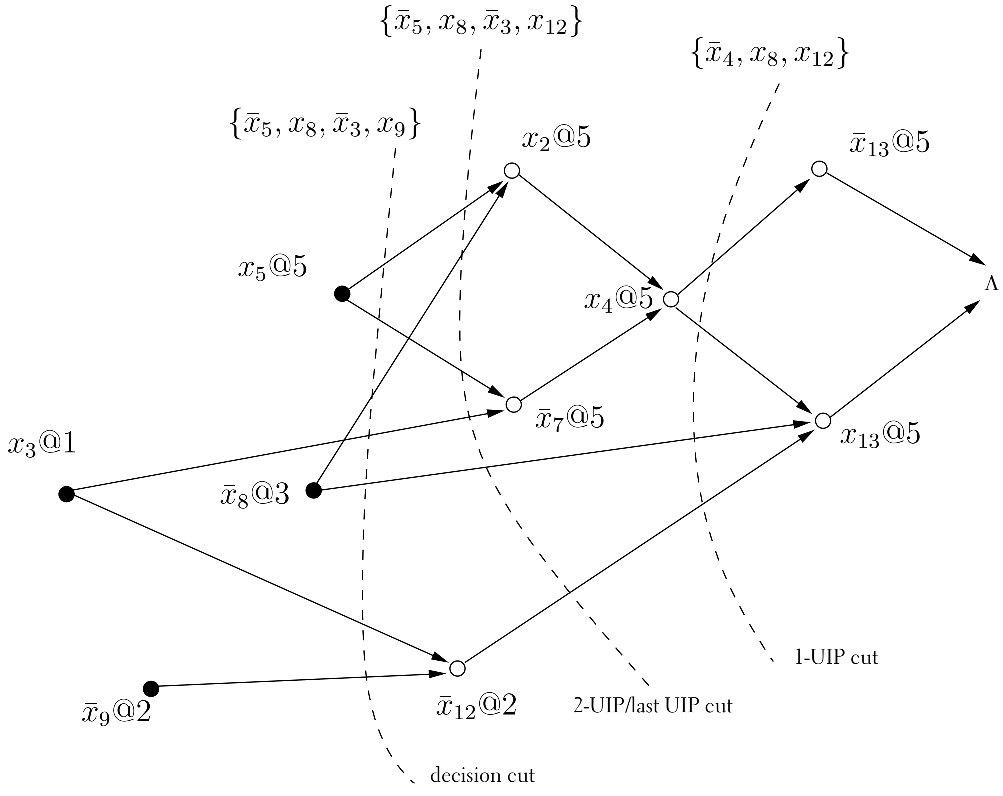


Figure 3.5: Example of a conflict graph, and three possible conflict cuts

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) [90]. A UIP in a conflict graph is a node u on the maximal 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, since 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 maximal 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 maximal 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 [137], showing the robustness of the 1-UIP scheme in practice.

Example 3.7 *Recall the conflict graph in Figure 3.5. 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 maximal 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 aim to follow [18], while trying to make the definition as precise as possible. A clause learning proof (or CL proof) induced by a learning scheme $\mathcal{S}$ is constructed by applying branching and the unit clause rule, using $\mathcal{S}$ 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* [18], 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, with leaf nodes labeled with the conflict graph, the conflict cut associated with the particular conflict, 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.

Any CL proof can be easily checked in polynomial time. The label at an arbitrary leaf node describes the generated conflict clause (by the cut in the graph). From the backjump level and the conflict graph it is easy to check that backjumping has been done correctly, i.e., by checking that either (i) the backjump level is one less than the maximal decision level (simple backtracking is used, or (ii) the conflict cut contains a UIP and that the maximal decision level of the other literals in the cut is equal to the backjump level. Finally, when the whole tree has been checked, it is easy to verify that the proof has been completed by applying UP on the known clauses.

It is worth noticing that, although simple backtracking and conflict-driven backjumping are quite different, CL can simulate DPLL even when restricting to UIP learning and conflict-driven backjumping. Intuitively, this is based on the fact that we can define a learning scheme (the *decision scheme*) which always learns a clause consisting exactly of the negations of all the decision literals in the conflict graph (see Figure 3.5 for an example of a decision cut). However, the opposite is not true. 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 [18]: CL can provide exponentially shorter proofs than T-RES even if no restarts are allowed. Thus we have the following corollary.

Corollary 3.2 (of Theorem 1 in [18]) 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 subformulas. 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 [60].

Beame et al. [18] 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 3.2 ([18]) RES *and* CL-- *with unlimited restarts and any non-redundant learning scheme are polynomially equivalent.*

We note that the proof of this theorem in [18] 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 clause learning DPLL implementations, in which unit propagation is applied immediately whenever possible.

3.5 CORRESPONDENCE BETWEEN A CIRCUIT AND ITS CNF TRANSLATION

A key element in this work is the tight correspondence between a constrained Boolean circuit $\mathcal{C}^\tau$ and its CNF translation $\mathsf{cnf}(\mathcal{C}^\tau)$. In more detail, when considering structural properties of variables in the CNF formula $\mathsf{cnf}(\mathcal{C}^\tau)$ resulting from the translation cnf of a constrained Boolean circuit $\mathcal{C}^\tau$, the properties are determined by $\mathcal{C}^\tau$ in which gates reflect one-to-one with the CNF variables of $\mathsf{cnf}(\mathcal{C}^\tau)$. For example, an *input variable* is a variable that corresponds to an input gate 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 2.1 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 [67], using a unit propagation equivalent characterization of Boolean constraint propagation as deduction rules for circuits [69]. Basically, such circuit level Boolean constraint propagation can set a value on a gate if and only if unit propagation can set a value on the corresponding Boolean variable in the CNF translation. For example, consider the gate $g := \text{AND}(g_1, g_2)$ and its CNF translation $\{\{\bar{g}, g_1\}, \{\bar{g}, g_2\}, \{g, \bar{g}_1, \bar{g}_2\}\}$. Now, for example, g can be assigned $\mathbf{f}$ if we have either $\langle g_1, \mathbf{f}\rangle$ or $\langle g_2, \mathbf{f}\rangle$ by the semantics of AND. On the CNF level, we can equivalently derive $\langle g, \mathbf{f}\rangle$ from $\langle g_i, \mathbf{f}\rangle$ by the clause $\{\bar{g}, g_i\}$ using the unit clause rule. 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., [88, 69, 79, 43, 86, 129]. 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. Note that circuit level solvers can also exploit additional propagation based on the concept of *don't cares* (for more, see [113, 52, 129]). In this work, however, we concentrate on applying unit propagation as the sole propagation mechanism, as typical especially in clause learning CNF level SAT solvers.

3.6 RESTRICTING BRANCHING IN DPLL TO INPUTS

In SAT based approaches to structured problems such as bounded model checking (of both hardware and software) and automated planning, the CNF encoding is often derived from a transition relation, where the behaviour of the underlying system is dependent on the *input*—initial state, nondeterministic choices, et cetera—of the system. Experimental case studies in specific problem domains [46, 126, 45] have shown that in some cases, SAT solvers benefit from restricting the variables the solver is allowed to branch on so called *input variables*, corresponding to the input of the underlying system. In the Boolean circuit encoding $\langle \mathcal{C}, \tau\rangle$ of such a structural problem, the input is represented by the set of input gates of the circuit, $\mathsf{inputs}(\mathcal{C})$. Since the circuit can be evaluated when all 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 $|\mathsf{inputs}(\mathcal{C})| << |G|$.

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 [67] 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 3.3 ([67]) $\mathsf{DPLL}_{\mathsf{inputs}}$ *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 [67]. In Section 5 we complement this theoretical study by providing an experimental evaluation of the effect of structure-based branching restrictions. In addition to investigating the effect of input-restricted branching—which is solely investigated often in the literature [46, 126, 45]—here the idea is to also study the effect of allowing branching in a controlled way on non–input gates based on properties imposed by the structure of the Boolean circuit at hand.

4 RESTRICTED BRANCHING AND PROOF COMPLEXITY

We will now consider the relative proo 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 Figure 4.1. 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 $*$ are due to trivial subsumption. The new results, detailed in the following, are represented by dashed arrows. Disregarding transitivity of the results, missing arrows represent questions which are open to the best of our knowledge.

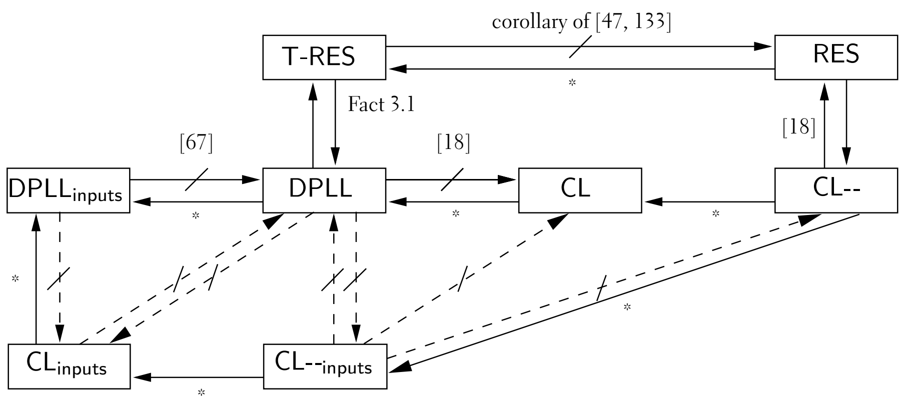


Figure 4.1: A refined relative efficiency hierarchy for the proof systems considered in this report.

The main result is characterized by the following theorem.

Theorem 4.1 DPLL *and* CL--$_{\text{inputs}}$ *(with or without restarts) are incomparable.*

This is a direct corollary of the forthcoming Lemmas 4.1 and 4.3. Thus we get the following as a direct corollary.

Corollary 4.1 CL--$_{\text{inputs}}$ *with unlimited restarts cannot polynomially simulate* CL.

We now proceed by proving Theorem 4.1 in two parts. First we show by a simple argument why DPLL cannot simulate CL$_{\text{inputs}}$. We then discuss further the difference between CL$_{\text{inputs}}$ and DPLL$_{\text{inputs}}$ by exhibiting an example of a family of Boolean circuits on which CL$_{\text{inputs}}$ *can* simulate CL, while 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 [67] on the power of DPLL$_{\text{inputs}}$ with respect to DPLL cannot be directly adopted for proving the analogous result for CL$_{\text{inputs}}$.

Although 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, 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 [131].

4.1 DPLL CANNOT SIMULATE $\mathsf{CL}_{\mathsf{inputs}}$

We now prove that DPLL cannot simulate $\mathsf{CL}_{\mathsf{inputs}}$.

Lemma 4.1 *There is an infinite family* $\{\mathcal{C}_n^\tau\}$ *of constrained Boolean circuits for which* DPLL *has exponentially longer minimal proofs than* $\mathsf{CL}_{\mathsf{inputs}}$.

Proof. Take any infinite family $\{F_n\}$ of CNF formulas that is a witness of Corollary 3.2 stating that DPLL cannot simulate CL. Define the family of Boolean circuits $\{\mathsf{circuit}(F) \mid F \in \{F_n\}\}$. Unit propagation on $\mathsf{cnf}(\mathsf{circuit}(F))$ without branching corresponds to the result of unit propagation on F without branching. Thus DPLL 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 3.2. □

As a direct corollary, we have

Corollary 4.2 *Neither* DPLL *nor* $\mathsf{DPLL}_{\mathsf{inputs}}$ *can polynomially simulate* $\mathsf{CL}_{\mathsf{inputs}}$.

4.2 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 $\{\text{XOR-UNSAT}_n\}$ 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 $\text{XOR-UNSAT}_n := \text{UNSAT} \cup \langle \text{XOR}_n^a \cup \text{XOR}_n^b, \emptyset\rangle$ consists of two parts:

(i) the constant size circuit

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

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

$$\text{XOR}_n^\rho := \{\rho := \text{ODD}(x_{1,1}^\rho, x_{1,2}^\rho)\} \cup \bigcup_{i=1}^{n-1} \bigcup_{j=1}^{i+1} \{x_{i,j}^\rho := \text{ODD}(x_{i+1,j}^\rho, x_{i+1,i+2}^\rho)\}.$$

The circuit XOR-UNSAT_2 is shown in Figure 4.2. Now, since UP will result in a conflict in the UNSAT subcircuit for any value of gate a, XOR-UNSAT_n yields a trivial (constant length) proof in DPLL. It is also easy to see that minimal length proofs of XOR-UNSAT_n are exponential with respect to n in $\mathsf{DPLL}_{\mathsf{inputs}}$. Due to the structure of XOR_n, in order to propagate a value for the gate a or b, $\mathsf{DPLL}_{\mathsf{inputs}}$ has to branch on all of the inputs in the corresponding XOR_n^ρ. With the backtracking process of DPLL this implies that minimal length $\mathsf{DPLL}_{\mathsf{inputs}}$ proofs of XOR-UNSAT_n are exponential with respect to n.

However, $\mathsf{CL}_{\mathsf{inputs}}$ can produce linear length proofs by simulating CL on the family. In the following we will say that CL (or DPLL) branches according to a sequence of assignments $(x_1 = v, \ldots)$, 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, and so forth. Now, let $\mathsf{CL}_{\mathsf{inputs}}$ branch according to the sequence $(x^a_{n,1} = \mathbf{f}, \ldots, x^a_{n,n} = \mathbf{f})$. After this, UP cannot still propagate any values. Then branch with $x^a_{n,n+1} = \mathbf{f}$. Now UP sets values for all $x^a_{i,j}$, without a conflict. The values for $x^a_{1,1}$ and $x^a_{1,2}$ propagate a value for a, which then propagates a conflict at a gate in UNSAT. Notice that $x^a_{1,1}$ and $x^a_{1,2}$ are the *only* reasons for the value of a. In *any* conflict graph associated with the branching sequence $(x^a_{n,1} = \mathbf{f}, \ldots, x^a_{n,n+1} = \mathbf{f})$, a is an 1-UIP, and, furthermore, constitutes a reason for the conflict on its own. Hence $\mathsf{CL}_{\mathsf{inputs}}$ can learn as a unit clause the opposite value of a, and backjump to decision level zero. This opposite value will then propagate a contradiction without branching, and $\mathsf{CL}_{\mathsf{inputs}}$ terminates.

It is interesting to notice how $\mathsf{CL}_{\mathsf{inputs}}$ can branch on $(x^a_{n,1} = \mathbf{f}, \ldots, x^a_{n,n+1} = \mathbf{f})$ and still avoid backtracking on these decisions since there is the *bottleneck* at gate a due to the construction of XOR-UNSAT$_n$. This shows the power 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 subformulas. With this intuition, it is evident that the results in [67] on the power of $\mathsf{DPLL}_{\mathsf{inputs}}$ with respect to DPLL cannot be directly adopted for proving the analogous result for $\mathsf{CL}_{\mathsf{inputs}}$.

4.3 CL--$_{\mathsf{inputs}}$ CANNOT SIMULATE DPLL

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

4.3.1 Redundant Gates

We will apply the concept of *redundant gates in constrained Boolean circuits*. For the following, a gate g in a constrained Boolean circuit $\langle\langle G, E\rangle, \tau\rangle$ is a

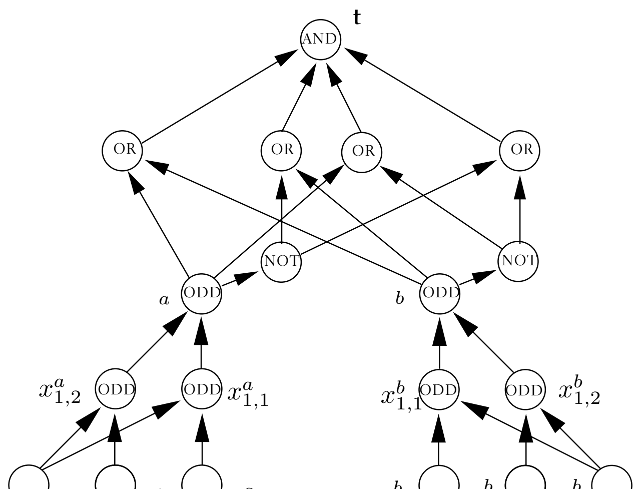


Figure 4.2: XOR-UNSAT$_n$ for $n = 2$

descendant of another gate g', if there is a path from g to g' in $\langle G, E\rangle$.

Definition 4.1 *A gate g in a constrained Boolean circuit $\langle\langle G, E\rangle, \tau\rangle$ is* redundant *if*

(i) g is unconstrained, and

(ii) g is not a descendant of any constrained gate g' in $\langle G, E\rangle$.

We will assume that circuits do not contain redundant input gates; such inputs can always be assigned an arbitrary truth value without affecting satisfiability.

Lemma 4.2 *Let $\mathcal{C}^\tau = \langle\langle G, E\rangle, \tau\rangle$ be an arbitrary constrained Boolean circuit. Considering $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ on input $\mathsf{cnf}(\mathcal{C}^\tau)$, redundant gates do not occur in any conflict graph at any stage of $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ whether or not restarts are allowed.*

Proof. Pick an arbitrary constrained Boolean circuit $\mathcal{C}^\tau$. The cases 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 on the input $\mathcal{C}^\tau$. Consider the $(m + 1)$th conflict. We prove by induction on the structure of $\mathcal{C}^\tau$ that no redundant gates occur in the conflict graph at the $(m + 1)$th conflict. The base case, considering a subcircuit with $n = 1$ gates, is trivial. Assume that the claim holds for all subcircuits with at most n gates. Let $\mathcal{C}^\tau_{n+1}$ be any subcircuit of $\mathcal{C}^\tau$ induced by a set G_{n+1} of $n + 1$ gates. Remove an arbitrary output gate $g := f(g_1, \ldots, g_k)$ from $\mathcal{C}^\tau_{n+1}$ to obtain a subcircuit induced by $G_{n+1} \setminus \{g\}$ with n gates. Such a g cannot be an input gate, since else it would not be connected to the rest of the circuit $\mathcal{C}^\tau$. Thus g is not branchable.

The case that g is not redundant is trivial. Now assume that g is redundant. Since there are no known learned clauses containing redundant gates before the $(m + 1)$th conflict, the only way to set a value for g is by UP from values set on (a subset of) $\{g_1, \ldots, g_k\}$. Any value for each g_i can be the result of UP on values for $G_{n+1} \setminus \{g\}$, or of branching in the case g_i is an input gate. For example, consider the case $g := \text{OR}(g_1, g_2)$. If g_1 has the value $\mathbf{t}$, g is propagated the value $\mathbf{t}$. After this, the value of g cannot propagate a value for g_2, nor can any value of g_2 propagate $\mathbf{f}$ for g. Other cases are similar. Thus the value of g cannot be used in propagating a value for any gate in $\mathcal{C}^\tau_{n+1}$, and therefore g cannot occur in any conflict graph for $\mathsf{CL}\text{--}_{\mathsf{inputs}}$. $\square$

From the proof of Lemma 4.2 it also follows that redundant gates can be removed from any constrained Boolean circuit, since such gates cannot contribute to any conflict, and thus cannot either have an effect on the satisfiability of any circuit.

4.3.2 Cook's Extension for a Polynomial Length Proof of PHP^{n+1}_n

Although redundant gates can be removed from any constrained Boolean circuit without affecting its satisfiability, they may have an effect on the length of minimal proofs. Cook [31] gives a way of introducing a polynomial number of clauses which can be interpreted as redundant gates to $\mathsf{circuit}(\mathrm{PHP}^{n+1}_n)$

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

$$\mathrm{EXT}^l := \bigcup_{i=1}^{l-1} \bigcup_{j=1}^{l-2} \{e_{i,j}^{l-1} := \mathrm{OR}(e_{i,j}^l, o_{i,j}^{l-1}),\ o_{i,j}^{l-1} := \mathrm{AND}(e_{i,l-1}^l, e_{l,j}^l)\},$$

and each $e_{i,j}^{n+1}$ is associated with the variable $p_{i,j}$ in PHP_n^{n+1}. A part of EXT_n is illustrated in Figure 4.3. The output gates of EXT_n are $e_{1,1}^2$ and $e_{2,1}^2$.

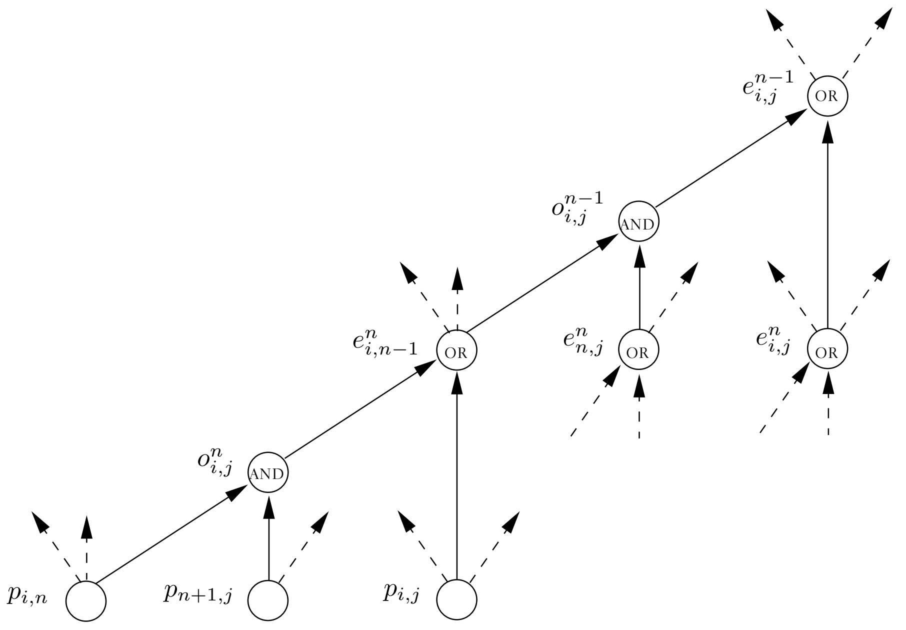


Figure 4.3: Part of Cook's extension EXT_n to PHP_n^{n+1} as a circuit

Due to the result in [31] we immediately have a polynomial length RES proof $\pi = (C_1, \ldots, C_m = \emptyset)$ of $\mathsf{cnf}(\mathsf{circuit}(\mathrm{PHP}_n^{n+1}) \cup \langle \mathrm{EXT}_n, \emptyset\rangle)$. Intuitively, EXT^l allows reducing PHP_l^{l+1} to PHP_{l-1}^l with a polynomial number of resolution steps. However, in [31] such a proof is not given explicitly, so we present one here.

The RES proof consists of four components, out of which the three first 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 $i = 1, \ldots, l+1$ and $j = 1, \ldots, l$, using the clauses in the CNF translation of $e_{i,j}^{l-1} := \mathrm{OR}(e_{i,j}^l, o_{i,j}^{l-1})$ and $o_{i,j}^{l-1} := \mathrm{AND}(e_{i,l-1}^l, e_{l,j}^l)$.

2. Derive the long clause $\{e_{i,1}^{l-1}, \ldots, e_{i,l-2}^{l-1}\}$ from $\{e_{i,1}^l, \ldots, e_{i,l-1}^l\}$ for each $i = 1, \ldots, l-1$.

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^2_1.

We will describe these steps now in more detail.

1. For each $e^{l-1}_{i,j} := \textsc{or}(e^l_{i,j}, o^{l-1}_{i,j})$ we have the clauses

$$\{\neg e^{l-1}_{i,j}, e^l_{i,j}, o^{l-1}_{i,j}\}, \{e^{l-1}_{i,j}, \neg e^l_{i,j}\}, \{e^{l-1}_{i,j}, \neg o^{l-1}_{i,j}\},$$

   and for each $o^{l-1}_{i,j} := \textsc{and}(e^l_{i,l-1}, e^l_{l,j})$ the clauses

$$\{o^{l-1}_{i,j}, \neg e^l_{i,l-1}, \neg e^l_{l,j}\}, \{\neg o^{l-1}_{i,j}, e^l_{i,l-1}\}, \{\neg o^{l-1}_{i,j}, e^l_{l,j}\}.$$

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

$$\{\neg e^{l-1}_{i,j}, e^l_{i,j}, e^l_{i,l-1}\}, \{\neg e^{l-1}_{i,j}, e^l_{i,j}, e^l_{l,j}\}, \{e^{l-1}_{i,j}, \neg e^l_{i,l-1}, \neg e^l_{l,j}\}.$$

2. The derivation is described in Figure 4.4. Notice that, at each step, the variable resolved upon is underlined. Recall that $\{e^{n+1}_{i,1}, \ldots, e^{n+1}_{i,n}\}$ is the clause $\{p_{i,1}, \ldots, p_{i,n}\}$ in PHP^{n+1}_n.

3. Figure 4.5 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 Figures 4.4 and 4.5 from $l = n+1$ to $l = 3$, one can thus derive the clauses $\{e^2_{1,1}\}$, $\{e^2_{2,1}\}$, and $\{\neg e^2_{1,1}, \neg e^2_{2,1}\}$, using which it is trivial to derive the empty clause in two resolution steps.

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 Figure 4.4. Hence the end result is not a T-RES proof.

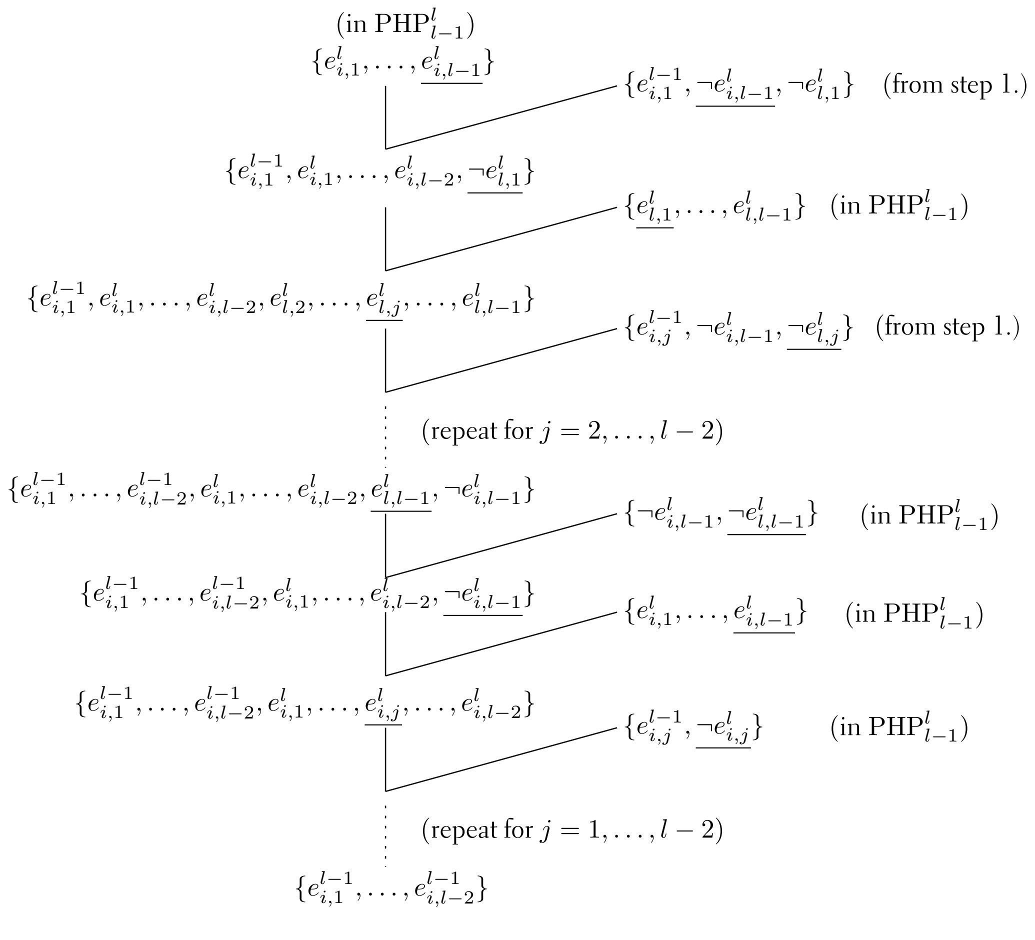

(in PHP^l_{l-1})
$\{e^l_{i,1}, \ldots, \underline{e^l_{i,l-1}}\}$
$\{e^{l-1}_{i,1}, \underline{\neg e^l_{i,l-1}}, \neg e^l_{l,1}\}$ (from step 1.)
$\{e^{l-1}_{i,1}, e^l_{i,1}, \ldots, e^l_{i,l-2}, \underline{\neg e^l_{l,1}}\}$
$\{\underline{e^l_{l,1}}, \ldots, e^l_{l,l-1}\}$ (in PHP^l_{l-1})
$\{e^{l-1}_{i,1},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

Figure 4.4: How to derive $\{e^{l-1}_{i,1}, \ldots, e^{l-1}_{i,l-2}\}$ in a polynomial number of resolution steps using Cook's extension for PHP^{n+1}_n

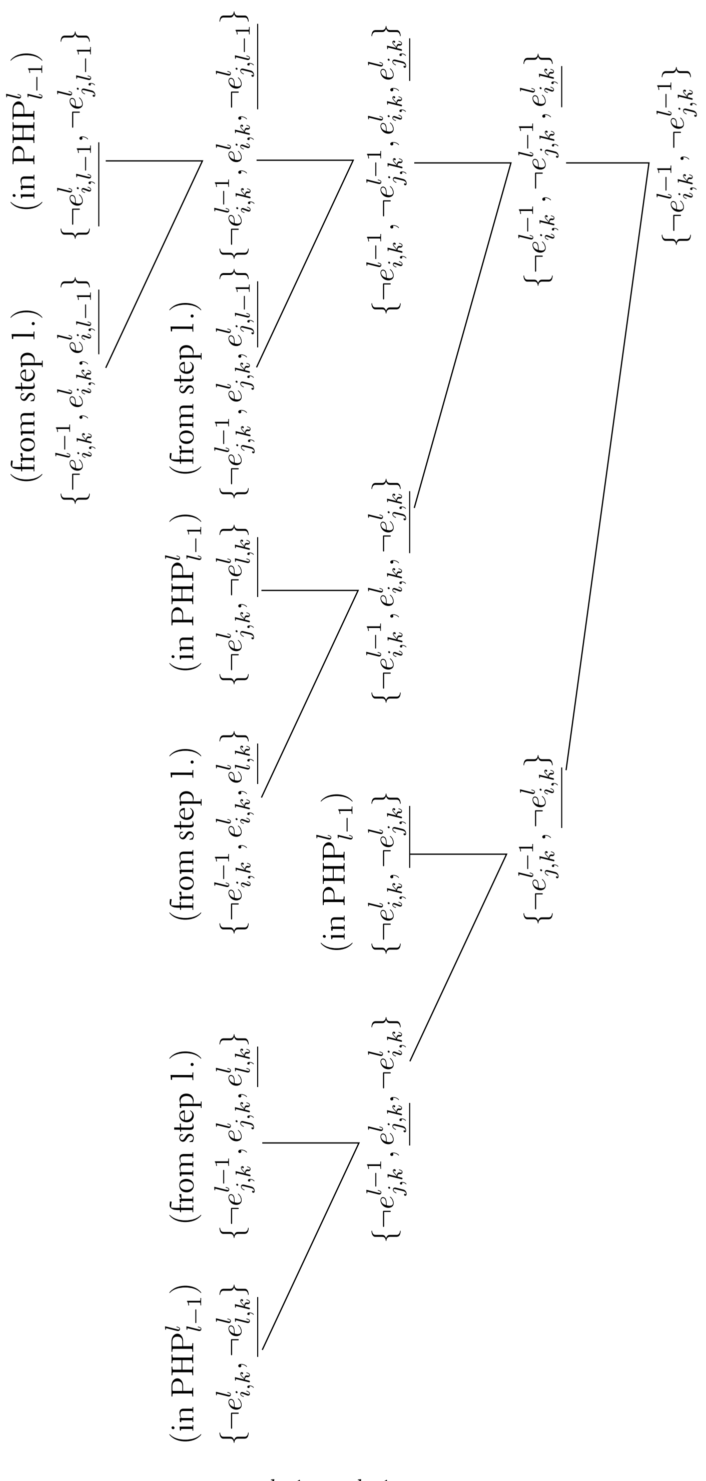

(in PHP^l_{l-1}) $\{\neg e^l_{i,k}, \neg e^l_{l,k}\}$
(from step 1.) $\{\neg e^{l-1}_{j,k}, e^l_{j,k}, e^l_{l,k}\}$
$\{\neg e^{l-1}_{j,k}, e^l_{j,k}, \neg e^l_{i,k}\}$
(in PHP^l_{l-1}) $\{\neg e^l_{i,k}, \neg e^l_{j,k}\}$
$\{\neg e^{l-1}_{j,k}, \neg e^l_{i,k}\}$
(from step 1.) $\{\neg e^{l-1}_{i,k},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

Figure 4.5: 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

4.3.3 The Separation

Using the above-described polynomial length RES proof $\pi = (C_1, C_2, \ldots, C_m = \emptyset)$ for the extended PHP_n^{n+1}, we define the circuit construct

$$\begin{aligned}\mathrm{E}(\pi) \;:=\; & \bigcup_{i=2}^{m-1} \{h_i := \mathrm{AND}(g_{C_i}, h_{i-1})\} \cup \\ & \bigcup_{i=1}^{m-1} \{g_{C_i} := \mathrm{OR}(g_{l_{i,1}}, \ldots, g_{l_{i,k_i}}) \mid C_i = \{l_{i,1}, \ldots, l_{i,k_i}\}\} \cup \\ & \bigcup_{i=1}^{m-1} \{g_{\bar{x}} := \mathrm{NOT}(g_x) \mid x \in \mathsf{vars}^-(C_i)\},\end{aligned}$$

where h_1 is the gate g_{C_1}. The construct $\mathrm{E}(\pi)$ is illustrated in Figure 4.6. In the figure the triangular shapes C_i stand for the circuit representation of the clause C_i in π.

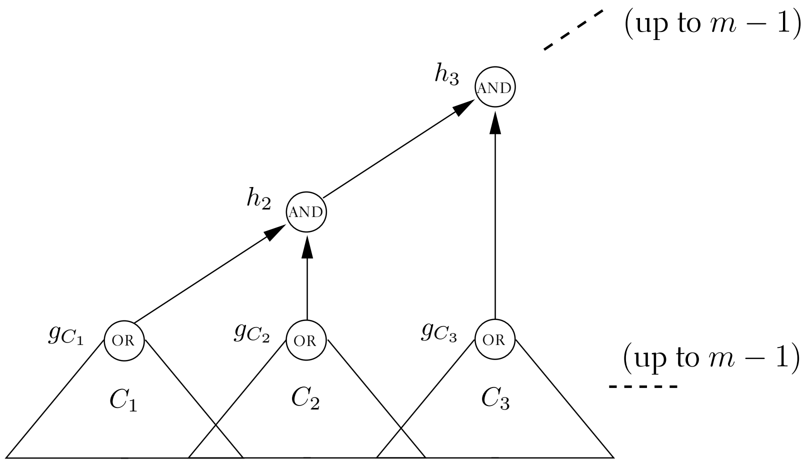


Figure 4.6: The construct $\mathrm{E}(\pi)$ based on a polynomial length RES proof $\pi = (C_1, C_2, \ldots, C_m = \emptyset)$ of the extended PHP_n^{n+1}

This allows a simple polynomial length DPLL proof of $\mathsf{cnf}(\mathrm{EPHP}_n^{n+1})$, where

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

while there is no polynomial length proof of $\mathsf{cnf}(\mathrm{EPHP}_n^{n+1})$ in $\mathsf{CL\text{-}\text{-}_{inputs}}$. 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 $\mathsf{CL\text{-}\text{-}_{inputs}}$ cannot make use of the redundant gates of EXT_n and $\mathrm{E}(\pi)$.

Lemma 4.3 *For the infinite family $\{\mathrm{EPHP}_n^{n+1}\}$ of constrained Boolean circuits,* $\mathsf{CL\text{-}\text{-}_{inputs}}$ *with unlimited restarts has superpolynomially longer minimal proofs than* DPLL.

Proof. A polynomial length DPLL proof of EPHP_n^{n+1} is witnessed by the branching sequence $(h_1 = \mathbf{f}, h_2 = \mathbf{f}, \ldots, h_{m-1} = \mathbf{f})$, as detailed next. By induction on i, we will show that, if $h_1 = \mathbf{t}, \ldots, h_{i-1} = \mathbf{t}$, then branching with $h_i = \mathbf{f}$ results in a conflict by UP, and hence immediately setting $h_i = \mathbf{t}$.

The base case. The gate $h_1 = g_{C_1}$ represents the first clause C_1 in π, and 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 $h_1 = \mathbf{f}$ will result in a conflict after UP because the functional definition or the constraint of the gate g is violated. For example, if $g := \mathrm{OR}(g_1, g_2)$ and $C_1 = \{x_g, \bar{x}_{g_1}\}$, then $h_1 = g_{C_1} := \mathrm{OR}(g, \hat{g_1})$, $\hat{g_1} := \mathrm{NOT}(g_1)$, and the assignment $h_1 = \mathbf{f}$ 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 $h_{i'} = \mathbf{t}$ for all $1 \leq i' < i$. Recall that $h_i := \mathrm{AND}(g_{C_i}, h_{i-1})$. By branching with $h_i = \mathbf{f}$, UP sets $g_{C_i} = \mathbf{f}$ by the induction hypothesis. 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 UP as in the base case. Otherwise C_i has been derived from two clauses, $C_j = C'_j \cup \{x_g\}$ and $C_k = C'_k \cup \{\bar{x}_g\}$, in π for $1 \leq j, k < i$, by resolving on a variable x_g. By the induction hypothesis we have $h_j = \mathbf{t}$ and $h_k = \mathbf{t}$, and thus $g_{C_j} = \mathbf{t}$ and $g_{C_k} = \mathbf{t}$ by UP. 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 UP, implying that UP 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 $h_i = \mathbf{f}$, and $h_i = \mathbf{t}$ is set by backtracking.

Finally, since $C_m = \emptyset \in \pi$, there are unit clauses $C_j = \{x_g\}$ and $C_k = \{\bar{x}_g\}$ in π, where $1 \leq j, k < m$. W.l.o.g., assume $j < k$. By induction, at latest after branching with $h_k = \mathbf{f}$ and setting $h_k = \mathbf{t}$ 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. The result is a linear DPLL proof.

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 4.2). 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 3.1 and 3.2). □

Now Theorem 4.1 follows directly from Lemmas 4.1 and 4.3.

4.4 ADDITIONAL REMARKS

Closely related to Lemma 4.3 and the applied construction EPHP_n^{n+1}, we make the following additional remarks.

- Due to the fact that redundant gates do not occur in *any* conflict graph of $\mathsf{CL}\text{--}_{\mathsf{inputs}}$, Lemma 4.3 covers all clause learning schemes based on conflict cuts, including, for example, schemes which learn *multiple clauses* at each conflict [90].

- We use redundant gates in the EPHP_n^{n+1} construction for simplicity of the proof of Lemma 4.3; by a simple modification of EPHP_n^{n+1} one can construct as a witness for Lemma 4.3 a constrained circuit with no redundant gates and a single output as the only constrained gate. The basic idea, illustrated in Figure 4.7, is to make a small local change to the EPHP_n^{n+1} circuit. In more detail, introduce the OR of the output gates $e^2_{1,1}$ and $e^2_{2,1}$ in EXT_n. Now, introduce the OR of this gate and the output gate h_{m-1} of $\mathrm{E}(\pi)$. Then, introduce the OR of this gate and the gate $\mathrm{NOT}(h_{m-1})$. Finally, constrain the AND of this gate and the output gate of circuit(PHP_n^{n+1}) to **t**. The resulting circuit witnesses Lemma 4.3, since conflicting assignment cannot still be propagated to the gates in $\mathrm{E}(\pi)$ or $\mathrm{E}(\pi)$ precisely because the added structure is a tautology of the form $\bar{h}_{m-1} \vee (h_{m-1} \vee \phi)$, that is, evaluates always to **t**.

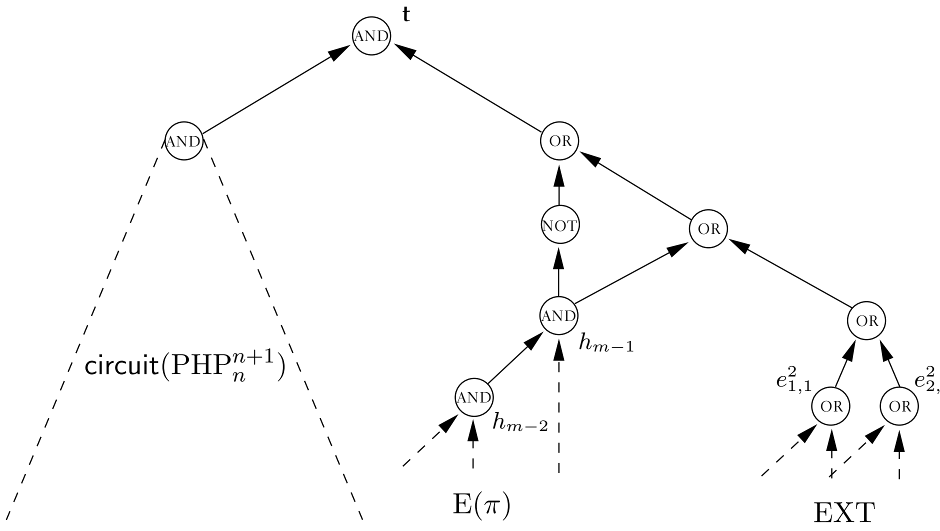


Figure 4.7: Local change to the EPHP_n^{n+1} circuit for removing redundancy of gates in $\mathrm{E}(\pi)$ and EXT_n

- Since redundant gates can be removed from constrained Boolean circuits without affecting the sets of satisfying assignments, such gates are typically removed in practice before the CNF translation by so called *cone-of-influence reduction* [69]. However, as witnessed by EPHP_n^{n+1} in Lemma 4.3, applying cone-of-influence can have a drastic negative effect on the minimal length proofs, since this will reduce EPHP_n^{n+1} to PHP_n^{n+1}.

- It is interesting to notice that DPLL solvers with full one-step lookahead can detect the small proofs of EPHP_n^{n+1} witnessed by the branching sequence ($h_1 = \mathbf{f}, h_2 = \mathbf{f}, \ldots, h_{n-1} = \mathbf{f}$). In particular, for each i, lookahead on $h_i = \mathbf{f}$ when having $h_j = \mathbf{t}$ for all $j < i$ in the branch will result in an immediate conflict using unit propagation, as detailed in the proof of Lemma 4.3.

- The Cook's extension (a variant of EXT_n) presented in [31] is motivated by investigations into the power of the *Extended Resolution proof system* defined by Tseitin [131]. Extended Resolution is the result of adding an *extension rule* to RES, which allows for adding *definitions* of the form $x \Leftrightarrow l_1 \wedge l_2$ (or, as a set of clauses, $\{\{x, \bar{l}_1, \bar{l}_2\}, \{\bar{x}, l_1\}, \{\bar{x}, l_2\}\}$) to the original 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 [76] for more details).

- The additional extension $E(\pi)$ applied above is motivated by a similar construction which can be used for simulating *Frege proofs* with their tree-like variants (see [76, Chapter 4]).

5 EXPERIMENTS

Complementing the theoretical results of the previous chapter, we evaluate the effect of structural branching restrictions on the behaviour of modern clause learning solver techniques. Before detailed discussion of the results, we describe the used Boolean circuit satisfiability benchmarks and the Boolean circuit front-end BCMinisat[1] applied in solving the instances.

5.1 BENCHMARKS

The benchmark set used in the experiments consists of instances from a number of real-life application domains, for which Boolean circuits offer a natural representation form. In selecting the benchmarks, the aim is to obtain a set of instances from multiple problem domains with varying structural properties. The selected benchmark set includes instances from verification of super-scalar processors [134], integer factorisation based on hardware multiplier designs [106], equivalence checking of hardware multipliers [66], bounded model checking (BMC) for deadlocks in asynchronous parallel systems modeled as labeled transition systems (LTSs) [70], and linear temporal logic (LTL) BMC of finite state systems with a linear encoding [81].

Verification of superscalar processors Boolean circuits encoding the problem of formally verifying the correctness of pipelined superscalar processors. The circuits are result of the translation from the logic of equality with uninterpreted functions to propositional logic presented in [134].

Bounded model checking for deadlocks in LTSs These circuits result from a translation scheme (using so called *interleaving* and *process semantics*) for BMC for deadlocks in a variety of asynchronous systems modeled as labeled transition systems [70].

Linear temporal logic BMC of finite state systems Linear size Boolean circuits encodings of BMC for finding bugs in finite state system designs violating properties specified in linear temporal logic (LTL) [81].

Integer factorization based on hardware multiplier designs These circuits encode the problem of finding factors of (both divisible and prime) numbers. The problem encodings are based on two hardware binary multiplier designs, the *adder tree* and *Braun* multipliers [25]. For a fixed n, both multipliers take as input two integers $\mathbf{a} = (a_1, \dots, a_n)$ and $\mathbf{b} = (b_1, \dots, b_n)$ as binary vectors, and output the product $\mathbf{o} = (o_1, \dots, o_{2n})$. Both designs consist of $\mathcal{O}(n^2)$ gates. However, the multipliers are structurally very unsimilar. The propagation delays (maximum of path lengths from inputs to outputs) are $\mathcal{O}(n)$ for Braun, and $\mathcal{O}(\log(n \log n))$ for adder tree. While Braun consists of a grid of full-adders, adder tree applies adders in a tree-like fashion, summing up

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

partial products. The circuits are obtained using the `genfacbm` benchmark generator [106].

Equivalence checking of hardware multipliers These circuits encode the problem of equivalence checking the results of the correct adder tree and Braun multipliers, as described in [66]. A Boolean circuit describing an instance of the equivalence checking problem for given n-bit adder tree (output bits $\mathbf{o}^{\mathrm{a}} = (o^{\mathrm{a}}_1, \ldots, o^{\mathrm{a}}_{2n})$) and Braun multipliers (output bits $\mathbf{o}^{\mathrm{b}} = (o^{\mathrm{b}}_1, \ldots, o^{\mathrm{b}}_{2n})$) is constructed as follows:

- The inputs of the multipliers are made equivalent by sharing the input gates $a_1, \ldots, a_n, b_1, \ldots, b_n$.
- Bit-wise equivalence of the outputs $\mathbf{o}^{\mathrm{a}}$ and $\mathbf{o}^{\mathrm{b}}$ is enforced by introducing gates $o^{\mathrm{eq}}_i := \textsc{equiv}(o^{\mathrm{a}}_i, o^{\mathrm{b}}_i)$ for $i = 1 \ldots 2n$.
- As a single output gate introduce **out** $:= \textsc{and}(o^{\mathrm{eq}}_1, \ldots, o^{\mathrm{eq}}_{2n})$.
- Constrain **out** to 0 (false).

Since the multiplier designs produce equivalent results for any two multiplicands, we arrive at unsatisfiable equivalence checking instances.

The set of Boolean circuit satisfiability benchmarks (a total of 38 instances, as detailed in Table 5.1) is available at

`http://www.tcs.hut.fi/~mjj/benchmarks/`.

For the experiments, we obtain a total of 570 CNF instances from these circuits as explained next.

5.2 SOLVING THE INSTANCES

For solving the Boolean circuit instances, we apply BCMinisat, which is a Boolean circuit front-end for the successful clause learning SAT solver Minisat[2] [41] (version 1.14). We use a farm of standard PCs with 2-GHz AMD 3200+ processors and 2 GBs of memory running Debian GNU Linux, with a timeout of 1 hour and a memory limit of 1 GB.

5.2.1 Simplification and CNF Translation in BCMinisat

BCMinisat accepts as input Boolean circuits with, in addition to the functions listed in Section 2.2, the following Boolean functions as gate types.

- $\textsc{equiv}(g_1, \ldots, g_n)$ evaluates to **t** if and only if (i) all $g_1, \ldots, g_n$ evaluate to **f**, or (ii) all $g_1, \ldots, g_n$ evaluate to **t**.

- $\textsc{even}(g_1, \ldots, g_n)$ evaluates to **t** if and only if even number of $g_1, \ldots, g_n$ evaluate to **t**.

- $\textsc{odd}(g_1, \ldots, g_n)$ evaluates to **t** if and only if odd number of $g_1, \ldots, g_n$ evaluate to **t**.

[2]`http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`

- $\text{CARD}_l^u(g_1, \ldots, g_n)$ evaluates to **t** if and only if at least l and at most u of $g_1, \ldots, g_n$ evaluate to **t**.

For handling these types, the BCMinisat front-end *normalizes* the circuit: these functions are *decomposed* by representing them with the gate types listed in Section 2.2 using auxiliary gates. The front-end also applies circuit-level preprocessing, including circuit-level Boolean propagation, substructure sharing, a circuit-level variant of the CNF-level *pure literal rule*, and *cone–of–influence reduction* [69] (redundant gates are removed) to the circuit. The resulting normalized and simplified circuit is translated into CNF applying the translation in Section 2.3. The only exception is that gates of form $g := \text{NOT}(g_1)$ are not translated; instead, $\neg\tilde{g}_1$ is substituted for $\tilde{g}$. BCMinisat feeds the resulting CNF formula and the input-restriction to Minisat for solving the instance. For each Boolean circuit satisfiability instance, we obtain 15 CNF instances by randomly permuting the CNF variable numbering with the `-permute_cnf` option of BCMinisat, making the total number of CNF formulas 570.

5.2.2 The Clause Learning CNF Solver Minisat

Minisat implements 1-UIP clause learning and a variation of the VSIDS heuristic [97]. After each conflict the heuristic values 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%. In the beginning, all heuristic values are set to zero. 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. Additionally, a restart strategy is applied.

We implemented the considered structural branching restrictions to BCMinisat, and modified Minisat so that its branching and heuristic can be restricted to a given set of variables. For insuring that restricting branching does not make decision making more time-consuming, we do not increment heuristic values for unbranchable variables, and additionally set the heuristic values of all branchable variables to one to make sure that time is not wasted on finding branchable variables even in the beginning of the search.

5.3 EXPERIMENT 1: EFFECT OF INPUT-RESTRICTED BRANCHING

Table 5.1 gives the minimum, median, and maximum number of decisions for BCMinisat and input-restricted BCMinisat (BCMinisat$_{\text{inputs}}$) for each Boolean circuit satisfiability benchmark instance. For the instances based on hardware multiplier designs, for which the number of unassigned input variables is 2% or less out of all unassigned variables, BCMinisat$_{\text{inputs}}$ shows an advantage over BCMinisat with respect to the number of decisions. However, for the hardware verification and BMC instances, the overall performance of BCMinisat$_{\text{inputs}}$ is much worse, with timeouts on all verification and half of the LTL BMC instances. The possible gains of input-restricted branching seems to correlate with a very low relative number of input variables. On the equivalence checking instances, we notice that the number of

Table 5.1: Minimum (**min**), median (**med**), and maximum (**max**) of number of decisions for BCMinisat and BCMinisat$_{\mathsf{inputs}}$, with number of timeouts in parenthesis. 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.

		Number of decisions								
		BCMinisat			BCMinisat$_{\mathsf{inputs}}$					
Instance	sat	min	med	max	min	med	max	#inputs	ud	bb
Super-scalar processor verification										
`fvp.2.0.3pipe.1`	no	**61531**	**384386**	**1225134**	- (15)	- (15)	- (15)	186 (8.2)	-	-
`fvp.2.0.3pipe_2_ooo.1`	no	**75962**	**184798**	**426489**	- (15)	- (15)	- (15)	305 (11.7)	-	-
`fvp.2.0.4pipe_1_ooo.1`	no	**188992**	**209048**	**271982**	- (15)	- (15)	- (15)	544 (10.4)	-	-
`fvp.2.0.4pipe_2_ooo.1`	no	**1033607**	**2094617**	**5241781**	- (15)	- (15)	- (15)	547 (9.8)	-	-
`fvp.2.0.5pipe_1_ooo.1`	no	**336281**	**746231**	**1838599**	- (15)	- (15)	- (15)	845 (8.9)	-	-
Equivalence checking hardware multipliers										
`eq-test.atree.braun.8`	no	180449	285665	339805	**65785**	**73834**	**82372**	16 (2.3)	88.5	0.02
`eq-test.atree.braun.9`	no	898917	1055511	1317785	**323688**	**385398**	**389890**	18 (2.0)	106.6	0.02
`eq-test.atree.braun.10`	no	3755375	4540598	5089443	**1428957**	**1590390**	**1787295**	20 (1.8)	127.9	0.01
Integer factorization										
`atree.sat.34.0`	yes	156733	228792	761620	**24820**	**208880**	**277896**	60 (0.6)	21.9	0.04
`atree.sat.36.50`	yes	**251218**	721474	937152	316590	**571533**	**788762**	64 (0.6)	18.4	0.04
`atree.sat.38.100`	yes	284980	1095192	- (1)	**190330**	**498092**	**1082729**	68 (0.6)	-	-
`atree.unsat.32.0`	no	141419	163508	180973	**123502**	**138797**	**162546**	57 (0.7)	15.3	0.04
`atree.unsat.34.50`	no	248371	287351	404418	**223130**	**244382**	**301464**	60 (0.6)	18.0	0.04
`atree.unsat.36.100`	no	527237	623889	915810	**431576**	**480469**	**578331**	64 (0.6)	19.4	0.03
`braun.sat.32.0`	yes	27480	82122	140150	**5675**	**81269**	**135093**	61 (2.2)	25.6	0.05
`braun.sat.34.50`	yes	**30717**	152224	353464	43924	**110614**	**223306**	65 (2.1)	25.3	0.05
`braun.sat.36.100`	yes	129771	447716	**589449**	**86134**	**374884**	752645	69 (2.0)	19.4	0.05
`braun.unsat.32.0`	no	107617	122550	156004	**96894**	**119437**	**150121**	60 (2.2)	10.4	0.06
`braun.unsat.34.50`	no	215624	263845	341855	**213199**	**258446**	**316819**	64 (2.0)	9.1	0.06
`braun.unsat.36.100`	no	**514725**	**623671**	807610	533575	640111	**674470**	68 (1.9)	8.9	0.06
BMC for deadlocks in LTSs										
`dp_12.i.k10`	no	**513935**	**639756**	**987595**	2497570	- (10)	- (10)	480 (16.0)	-	-
`key_4.p.k28`	no	**121552**	**147063**	**169386**	138361	184875	220107	967 (10.9)	3.7	0.53
`key_4.p.k37`	yes	56784	**321552**	**1549271**	**7574**	663152	- (1)	1507 (9.8)	-	-
`key_5.p.k29`	no	**193139**	**223867**	**310207**	230844	343255	405686	1212 (10.7)	3.9	0.54
`key_5.p.k37`	yes	104496	**421324**	**1540174**	**19027**	1041807	- (3)	1796 (9.8)	-	-
`mmgt_4.i.k15`	no	**210288**	**287599**	**457009**	582998	1105986	2170048	456 (10.9)	4.2	0.41
`q_1.i.k18`	no	**168156**	**353421**	**507246**	375493	929019	1349785	566 (13.1)	3.7	0.49
LTL BMC by linear encoding										
`1394-4-3.p1neg.k10`	no	141822	155295	164900	**138468**	**148545**	**156839**	1845 (5.6)	6.6	0.34
`1394-4-3.p1neg.k11`	yes	72988	128708	203647	**34619**	**55575**	**189434**	2023 (5.5)	9.0	0.32
`1394-5-2.p0neg.k13`	no	**125840**	**143928**	**158320**	146144	156527	186468	1940 (5.0)	6.7	0.32
`brp.ptimonegnv.k23`	no	**106338**	**130577**	**259025**	193839	302930	356313	461 (6.7)	4.1	0.28
`brp.ptimonegnv.k24`	yes	43013	96775	**162114**	**13699**	**74907**	260481	481 (6.7)	5.5	0.27
`csmacd.p0.k16`	no	**229192**	**316082**	**376280**	269520	341751	381248	1794 (2.9)	4.9	0.28
`dme3.ptimo.k61`	no	**314659**	**549686**	**1658757**	- (15)	- (15)	- (15)	6375 (26.3)	-	-
`dme3.ptimo.k62`	yes	**427100**	**688505**	**1545603**	- (15)	- (15)	- (15)	6506 (26.3)	-	-
`dme3.ptimonegnv.k58`	no	**324770**	**568864**	**962967**	- (15)	- (15)	- (15)	5982 (26.3)	-	-
`dme3.ptimonegnv.k59`	yes	**303921**	**480073**	**1136938**	- (15)	- (15)	- (15)	6113 (26.3)	-	-
`dme5.ptimo.k65`	no	**497190**	**735741**	**1839619**	- (15)	- (15)	- (15)	10750 (26.8)	-	-

decision for BCMinisat$_{\mathsf{inputs}}$ *is more than the brute-force upper bound, that is, the size of the search space.* For example, 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 most likely due to conflict clause forgetting; when forgetting a conflict clause C, the solver may have to re-examine the search space characterised as unsatisfiable by C.

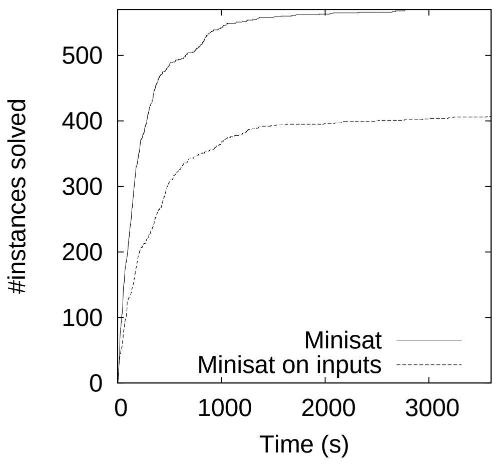


Figure 5.1: Cumulative number of solved instances

In Figure 5.1 we have a cumulative plot of the number of solved instances as a function of time, showing a drastic decrease in performance for the input-restricted branching Minisat. The effect of input-restricted branching varies depending on whether unsatisfiable or satisfiable instances are considered (Figure 5.2).

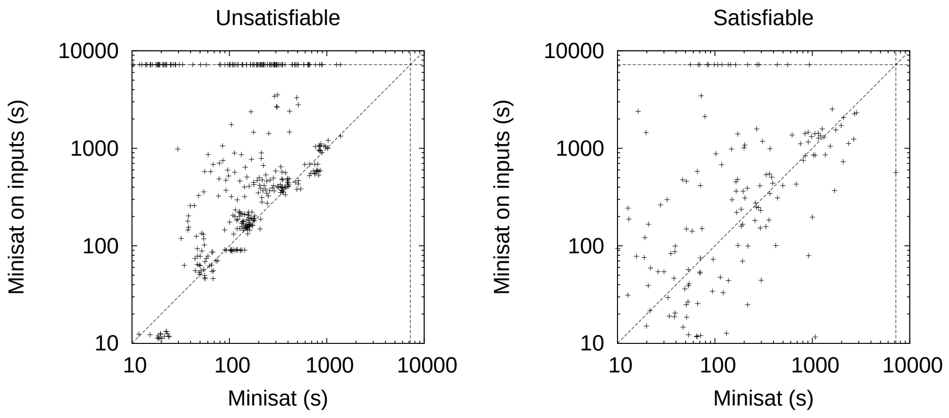


Figure 5.2: Running times on unsatisfiable (left) and satisfiable (right) instances

On unsatisfiable instances input-restriction results in a clear efficiency decrease, with timed out runs shown 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. A noticeable point is that, while BCMinisat$_{\mathsf{inputs}}$ makes less decisions, for example, on the equivalence checking instances, unrestricted BCMinisat is at least as efficient as BCMinisat$_{\mathsf{inputs}}$ when looking at running times. Interestingly, this is due to the fact that unrestricted BCMinisat often *manages more decisions per second* (Figure 5.3 over the set of CNF instances solved by both BCMinisat and BCMinisat$_{\mathsf{inputs}}$).

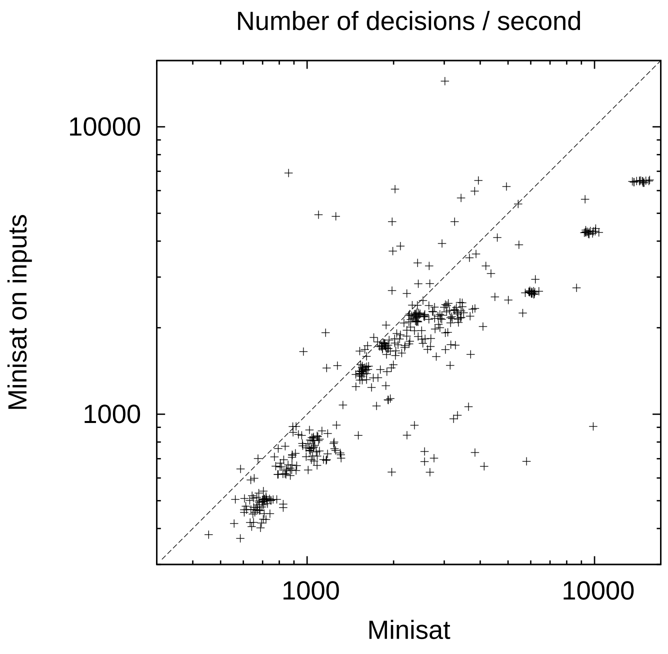


Figure 5.3: Number of decisions / second

We can make more interesting observations by looking at statistics over all instances solved by both BCMinisat and BCMinisat$_{\mathsf{inputs}}$. An important aspect in the effectiveness of clause learning are the lengths of learned clauses, i.e., the number of literals in the clauses. Since a conflict clause describes an unsatisfiable part of the search space, shorter conflict clauses are intuitively exponentially more effective than longer ones. In Figure 5.4 we have a comparison of the average lengths of learned clauses in the solved instances. With input-restricted branching the learned clauses are typically evidently longer. Longer learned clauses can also have a negative effect on the efficiency of the solver, since handling the clauses can take more time, for example, to propagate. This would partly explain the decrease in the number of decisions per time unit for input-restricted branching Minisat.

We also look at the maximal decision levels visited by BCMinisat and BCMinisat$_{\mathsf{inputs}}$ on the different instance families (Figure 5.5). The intuitive drop in the worst-case behaviour of Minisat resulting from input-restricted

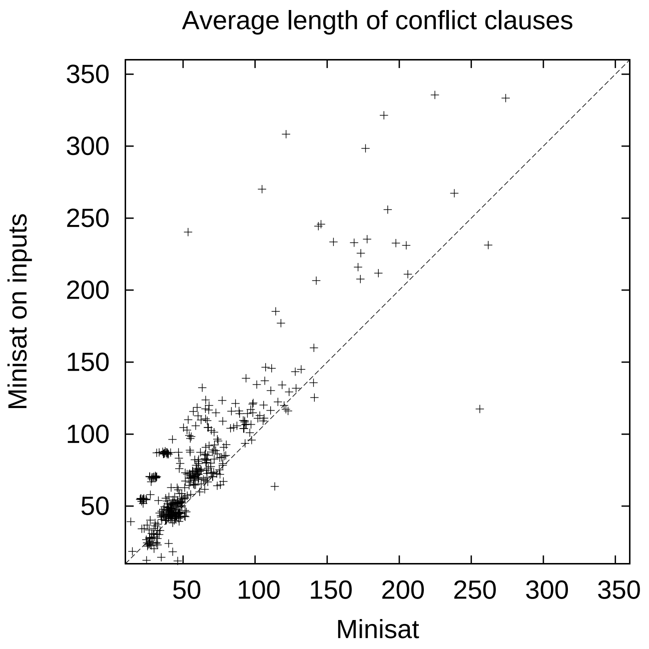


Figure 5.4: Average length of conflict clauses

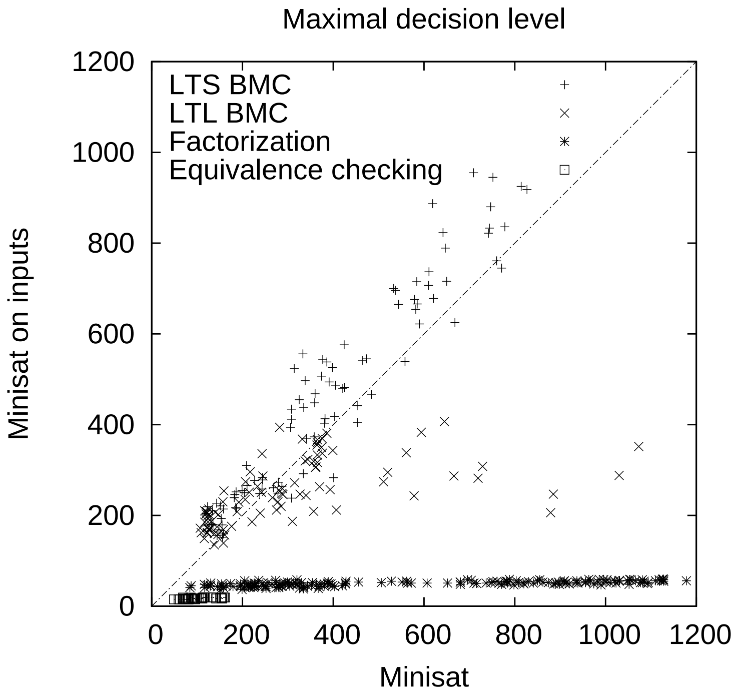


Figure 5.5: Maximal decision levels

branching is reflected in the maximal decision levels for the families based on multiplier designs, where the number of input variables is very low (see column #inputs in Table 5.1). For the LTS BMC instances, however, the decision levels are greater for the input-restricted branching solver, although the number of input variables is still only around 10% out of all unconstrained variables.

We also observe that the VSIDS heuristic might not work as intended with input-restricted branching. 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 5.1). For example, for `eq-test.atree.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 5.1)—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 when developing suitable heuristics and learning schemes.

5.4 EXPERIMENT 2: RELAXED STRUCTURAL BRANCHING RESTRICTIONS

In order to study whether the robustness of input-restricted branching can be improved while still branching on a subset of variables, we now apply controlled schemes for allowing branching additionally on CNF variables other than input variables based on structural properties of Boolean circuits. The general idea here is to allow—in addition to input variables—branching consistently on the best $p\%$ of unconstrained non-input variables according to criteria that are based on different aspects of the underlying circuit structure. Input variables are always included for assuring that Minisat remains complete under the restrictions.

For the following, let $\mathcal{C}^\tau$ be a simplified and normalized constrained circuit with the sets of unconstrained gates G, input gates $\mathsf{inputs}(\mathcal{C}^\tau)$, and output gates $\mathsf{outputs}(\mathcal{C}^\tau)$. For a gate $g := f(g_1, \ldots, g_n)$, the set of g's children is $\mathsf{children}(g) = \{g_1, \ldots, g_n\}$, and the set of g's parents is $\mathsf{parents}(g)$. For a gate $g \in G$, the *fanout* $\mathsf{fanout}(g)$ is the number of gates whose child g or $g' := \text{NOT}(g)$ is. The *degree* $\mathsf{degree}(g)$ is the sum of $\mathsf{fanout}(g)$ and the number of g's children. Additionally, let $\Delta^{\max}_{\mathsf{inputs}}(g)$ denote the length of the longest path under the child relation of $\mathcal{C}^\tau$ from g to any input gate. Here NOTs do not contribute to the length of the paths, since they are not translated. Similarly, $\Delta^{\max}_{\mathsf{outputs}}(g)$ stands for the length of the longest path under the parent relation of $\mathcal{C}^\tau$ from g to any output gate. The duals of $\Delta^{\max}_{\mathsf{inputs}}(g)$ and $\Delta^{\max}_{\mathsf{outputs}}(g)$ are $\Delta^{\min}_{\mathsf{inputs}}(g)$ and $\Delta^{\min}_{\mathsf{outputs}}(g)$, that is, denoting the lengths of the shortest paths from g to any input and output, respectively.

We will investigate the following criteria.

Random restriction (denoted by $\mathsf{rnd}(p)$**):** As a reference point for the other structural restrictions, we allow branching on $p\%$ of randomly chosen unconstrained non-input variables. Intuitively, this results in allowing branching evenly across the underlying circuit structure.

Fanout-based restriction $\mathsf{fan}(p)$**:** Here gates are ranked according to the values $\mathsf{fanout}(g)$, with the criterion that gates with large values are preferred. This is a generalization of the idea of restricting branching to gates g with $\mathsf{fanout}(g) > 1$ as suggested in the context of SAT-based ATPG [122].

Degree-based restriction $\mathsf{deg}(p)$: Here gates are ranked according to the values $\mathsf{degree}(g)$, with the criterion that gates with large values are preferred. The value $\mathsf{degree}(g)$ is closely related to the number of occurrences of the variable corresponding to gate g in the CNF translation of $\mathcal{C}^\tau$. Hence, this restriction is related to the counting based branching heuristics such as DLIS and MOMS, in which heuristic values are based on counting the number of occurrences of variables/literals [57].

Flow-based restriction $\mathsf{flow}(p)$: Here gates are ranked according to the values $\mathsf{flow}(g)$, as defined below, with the criterion that gates with large values are preferred.

$$\mathsf{flow}(g) = \begin{cases} \frac{1}{|\mathsf{outputs}(\mathcal{C}^\tau)|} & \text{if } g \in \mathsf{outputs}(\mathcal{C}^\tau) \\ \displaystyle\sum_{g' \in \mathsf{parents}(g)} \frac{\mathsf{flow}(g')}{|\mathsf{children}(g')|} & \text{otherwise} \end{cases}$$

In other words, we compute a total flow value for each gate by pouring a constant quantity of flow down from the output gates of the circuit. Notice that in the simplified and normalized circuit $\mathcal{C}^\tau$, the output gates are always constrained by τ. Here the intuitive idea is that, if a large total flow passes through a gate g, the gate is *globally* very connected with the constraints in τ, and thus g would have an important role in the satisfiability of the circuit.

Distance-based restrictions: Complementing the other restrictions based on the underlying structure of Boolean circuits, we also consider restricting branching based on the distances of gates from inputs and outputs.

- In $\mathsf{minmax} - \mathsf{dist}(p)$ gates are ranked according to the values

  $$\max\{\Delta^{\max}_{\mathsf{inputs}}(g), \Delta^{\max}_{\mathsf{outputs}}(g)\},$$

  with the criterion that gates with small values are preferred. Here the idea is to concentrate branching on variables that are close to both input and output variables.
- In $\mathsf{maxmin} - \mathsf{dist}(p)$ gates are ranked according to the values

  $$\min\{\Delta^{\min}_{\mathsf{inputs}}(g), \Delta^{\min}_{\mathsf{outputs}}(g)\},$$

  with the criterion that gates with large values are preferred. Here the idea is to concentrate branching on variables that are far from both input and output variables (the dual of $\mathsf{minmax} - \mathsf{dist}(p)$).

In selecting the $p\%$ of variables according to a particular criterion, ties are broken randomly from the set of variables having the *break value* of the criterion. For example, consider $\mathsf{fan}(p)$. Let k be the break value such that

$$100 \times |\{g \mid \mathsf{fanout}(g) \geq k\}|/|G| \geq p$$

and

$$100 \times |\{g \mid \mathsf{fanout}(g) \geq k+1\}|/|G| < p$$

hold. Now branching is allowed on all gates g with $\mathsf{fanout}(g) \geq k+1$ and additionally on a number of randomly chosen gates g with the break value $\mathsf{fanout}(g) = k$ so that the percentage p is reached.

We run BCMinisat with all the above-mentioned branching restrictions and values $p = 10, 20, 40, 60, 80$. The results as the cumulative number of solved instance are shown in Figure 5.6. First, as witnessed by the random restriction, by allowing branching additionally on non-input variables the robustness of Minisat increases gradually. Considering the structural restrictions, it is interesting to see that for the fanout and degree based restrictions only 20% additional branching variables are enough for the restrictions to reach a level of robustness very close to unrestricted branching Minisat. For the flow-based restriction, this holds from 40% on. It is very interesting to see that the choice of the structural criterion *does make a difference*: we observe that the distance-based restrictions result in very poor performance. In fact, the only restrictions on which Minisat solves all the CNF instances are $\mathsf{deg}(20)$, $\mathsf{deg}(40)$, and $\mathsf{flow}(40)$ (for example, Minisat without any restrictions on branching time outs on one instance out of the 570 CNF formulas, see Table 5.1). From these results we draw the conclusion that branching *can be restricted* even rather heavily without losing much of the robustness of a clause learning SAT solver on various instance families. On the other hand, at least the considered *structural restrictions do not seem to be beneficial in general* on their own, since none of them give notable gains compared to unrestricted branching with respect to running times.

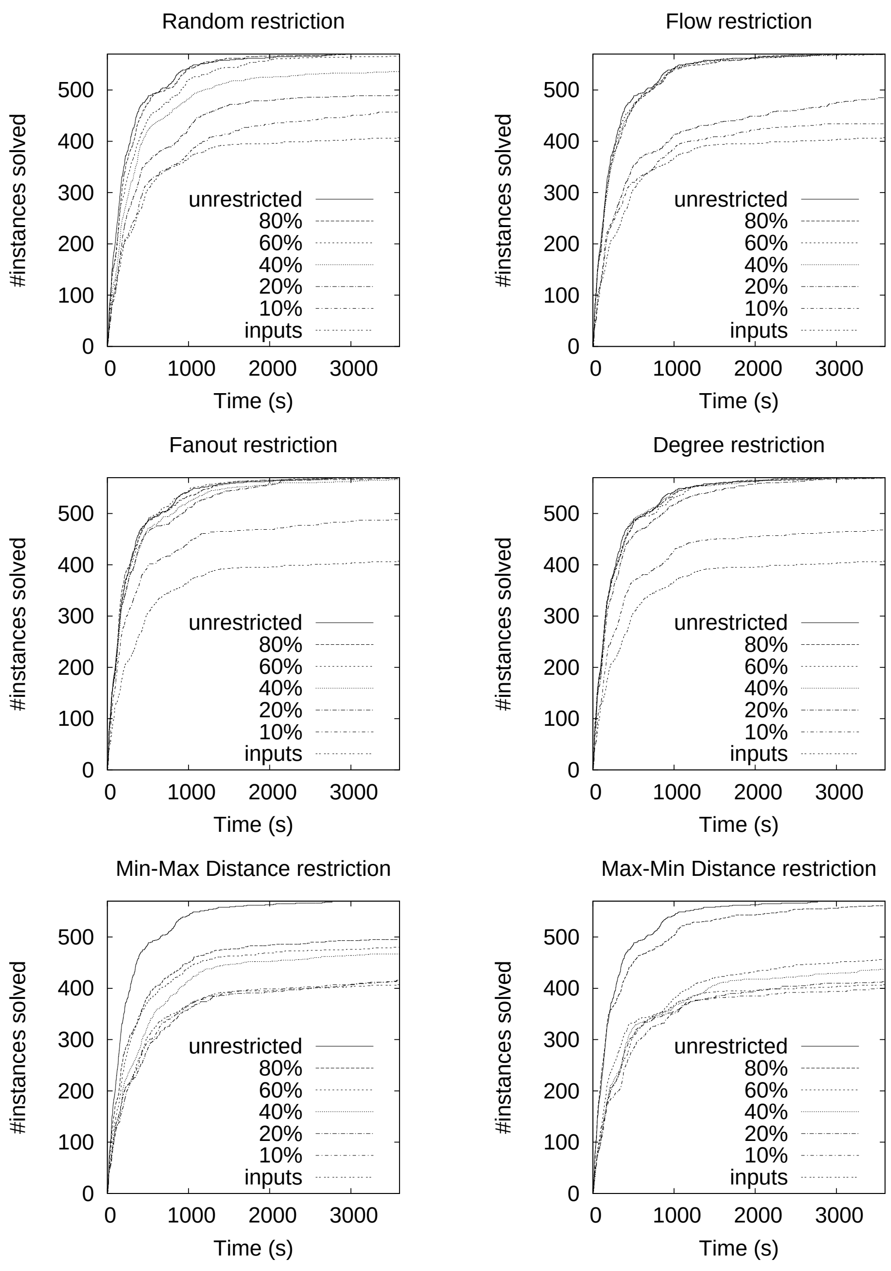


Figure 5.6: Cumulative number of solved instances for the structural branching restrictions

6 CONCLUSIONS

This work aims at contributing to the understanding of what type of search techniques yield increasingly robust SAT solving engines for industrial-scale structural combinatorial problems. The focus is on the effect of structure-based branching restrictions on the efficiency of modern SAT solving techniques. The work is well–motivated by the fact that, while techniques such as novel decision heuristics and clause learning have been the focus of much attention, the structural properties underlying CNF encodings of real-world problems have not been extensively studied from the view point of branching restrictions. Although branching plays a key role in search for satisfiability, there still is no general consensus on what type of structural properties (if any) reflect variables with high importance with respect to efficiency of search, and how such knowledge could be exploited in making SAT solvers more robust.

This work extends previous work on branching restrictions in DPLL based SAT solving procedures both on the theoretical and the practical level. With propositional proof complexity as the framework, we show that when branching is restricted to input variables in clause learning DPLL, the resulting underlying proof system weakens considerably; input-restricted branching clause learning DPLL and basic DPLL are polynomially incomparable. This holds even when input-restricted branching clause learning DPLL is allowed unlimited restarts and the ability to branch on variables with already assigned values. Thus we provide an answer to the question of the relative efficiency of input-restricted branching clause learning DPLL with respect to clause learning DPLL, which was posed as an open problem in [67]. This also implies that all implementations of clause learning DPLL, even with optimal heuristics, have the potential of suffering a notable efficiency decrease when input-restricted branching is applied.

The experiments confirm that, in general, input-restricted branching can cause a notable loss of robustness in a clause learning SAT solver. This is evident especially on unsatisfiable problem instances. We also show that input-restricted branching results in, for example, longer conflict clauses on the average, which in itself makes clause learning less effective and can also hinder the overall efficiency of the solver. However, by relaxing the input-restriction by allowing branching additionally on variables with particular underlying structural properties in a systematic fashion, we are able to show that branching can in fact be restricted quite heavily without making a clause learning solver notably less efficient. Moreover, the choice of the structural property on which such a relaxation is based on does make a difference.

6.1 TOPICS FOR FURTHER STUDY

Investigating each of the following topics could either strengthen the results in this work, or are interesting research topics closely related to this work.

- Theorem 4.1 states that there is a family $\{F_n\}$ of CNF formulas on which the minimal length proofs in input-restricted branching clause learning DPLL are superpolynomially longer than the ones in the basic DPLL with respect to the number of clauses in each F_n. Can this result be strengthened to exponential differences, that is, an *exponential separation*? This would require a family of circuits for which the sizes of the circuits grow linear in n.

- The experiments in this work show that, in general, many branching restrictions based on natural structural properties of circuit gates do not give notable gains. Can such, relatively small branching restrictions still be found, for example, by further analysis *combinations of* structural properties?

- If static branching restrictions do not increase efficiency on their own, as it seems by the experimental results in this work, one could investigate more complex, dynamic branching restrictions for structural problems. An interesting question is whether one can gain from integrating branching restrictions more tightly with clause learning.

- What is the exact power of clause learning DPLL without restarts? Can it polynomially simulate RES without further relaxations? A positive answer to this would further clarify the hierarchy shown in Figure 4.1.

- This work concentrates on branching restrictions for complete DPLL based SAT solvers. What about local search for structural problems; can structure and completeness-preserving branching restrictions, with additional propagation mechanisms, be applied in restricting the set of variables which to flip to the extend that local search for structural problems would become feasible? Input-restricted flipping has been considered to some extent in the literature [71, 103]. Could one do better with more sophisticated flip restrictions?

ACKNOWLEDGEMENTS

This report is a reprint of my Licentiate's thesis, resulting from my studies in the Laboratory for Theoretical Computer Science at Helsinki University of Technology during the years 2005–2007. The work has been done under the supervision of Professor Ilkka Niemel and under the guidance of D.Sc.(Tech.) Tommi Junttila. I thank them for many insights, fruitful discussions, and collaboration in my studies. The external reviewer of the thesis is Docent, D.Sc.(Tech.) Tomi Janhunen, whom I thank for many suggestions for improving the representation. In addition, I thank Lic.Sc.(Tech.) Emilia Oikarinen for help on figuring out Cook's easy exercise.

The following parties have provided financial support, for which I am grateful: Helsinki Graduate School in Computer Science and Engineering (HeCSE), Helsinki University of Technology, Academy of Finland (under the projects *Advanced Constraint Programming Techniques for Large Structured Problems* and *Applications of Rule-Based Constraint Programming*), Emil Aaltonen Foundation, and Finnish Foundation of Technology (TES).

BIBLIOGRAPHY

[1] Parosh Aziz Abdulla, Per Bjesse, and Niklas E n. Symbolic reachability analysis based on SAT-solvers. In Susanne Graf and Michael I. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2000.

[2] Dimitris Achlioptas, Paul Beame, and Michael Molloy. Exponential bounds for DPLL below the satisfiability threshold. In J. Ian Munro, editor, *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 139–140. SIAM, 2004.

[3] Dimitris Achlioptas, Paul Beame, and Michael S. O. Molloy. A sharp threshold in proof complexity yields lower bounds for satisfiability search. *Journal of Computer and System Sciences*, 68(2):238–268, 2004.

[4] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[5] Michael Alekhnovich. Mutilated chessboard problem is exponentially hard for resolution. *Theoretical Computer Science*, 310(1–3):513–525, 2004.

[6] Michael Alekhnovich, Edward A. Hirsch, and Dmitry Itsykson. Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. *Journal of Automated Reasoning*, 35(1–3):51–72, 2005.

[7] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing (STOC'02)*, pages 448–456. ACM, 2002.

[8] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. MINCE: A static global variable-ordering heuristic for SAT search and BDD manipulation. *Journal of Universal Computer Science*, 10(12):1562–1596, 2004.

[9] Anbulagan and John Slaney. Lookahead saturation with restriction for SAT. In Peter van Beek, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 727–731. Springer, 2005.

[10] Henrik Reif Andersen and Henrik Hulgaard. Boolean expression diagrams. *Information and Computation*, 179(2):194–212, 2002.

[11] Alessandro Armando, Luca Compagna, and Pierre Ganty. SAT-based model-checking of security protocols using planning graph analysis. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proceedings of the 2003 International Symposium of Formal Methods Europe (FME'03)*, volume 2805 of *Lecture Notes in Computer Science*, pages 875–893. Springer, 2003.

[12] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2006.

[13] Albert Atserias, Phokion G. Kolaitis, and Moshe Y. Vardi. Constraint propagation as a proof system. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2004.

[14] Fahiem Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'02)*, pages 613–619. AAAI Press, 2002.

[15] Paul Beame, Joseph C. Culberson, David G. Mitchell, and Cristopher Moore. The resolution complexity of random graph k-colorability. *Discrete Applied Mathematics*, 153(1–3):25–47, 2005.

[16] Paul Beame, Russell Impagliazzo, and Ashish Sabharwal. The resolution complexity of independent sets and vertex covers in random graphs. *Computational Complexity*, to appear.

[17] Paul Beame, Richard M. Karp, Toniann Pitassi, and Michael E. Saks. The efficiency of resolution and Davis–Putnam procedures. *SIAM Journal on Computing*, 31(4):1048–1075, 2002.

[18] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[19] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th Conference on Design Automation (DAC'99)*, pages 317–320. ACM Press, 1999.

[20] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5:5), 2006.

[21] Per Bjesse and Arne Bor lv. DAG-aware circuit compression for formal verification. In *2004 International Conference on Computer-Aided Design (ICCAD'04)*, pages 42–49. IEEE Computer Society / ACM, 2004.

[22] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys*, 38(4):Article 12, 2006.

[23] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *Journal of Automated Reasoning*, 35(1–3):265–293, 2005.

[24] Alfredo Braunstein, Marc M zard, and Riccardo Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2):201–226, 2005.

[25] Stephen Brown and Zvonko Vranesic. *Fundamentals of digital logic with VHDL design*. McGraw Hill, 2000.

[26] Randy E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[27] Claudio Castellini, Enrico Giunchiglia, and Armando Tacchella. SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147(1–2):85–117, 2003.

[28] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic logic and mechanical theorem proving*. Academic Press, 1973.

[29] Vašek Chv tal and Endre Szemer di. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, 1988.

[30] Stephen A. Cook. The complexity of theorem proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[31] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, 1976.

[32] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, 1979.

[33] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI'94)*, pages 1092–1097. AAAI Press, 1994.

[34] Stefan Dantchev and Soren Riis. "Planar" tautologies hard for resolution. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS'01)*, pages 220–229. IEEE Computer Society, 2001.

[35] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[36] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[37] Debapratim De, Abishek Kumarasubramanian, and Ramarathnam Venkatesan. Inversion attacks on secure hash functions using SAT solvers. In Jo o Marques-Silva and Karem A. Sakallah, editors, *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*. Springer, 2007.

[38] Thierry Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14:283–301, 1992.

[39] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[40] Rolf Drechsler and Bernd Becker. *Binary Decision Diagrams – Theory and Implementation*. Kluwer Academic Publishers, 1998.

[41] Niklas E n and Niklas S rensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Revised Selected Papers of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[42] Zhaohui Fu and Sharad Malik. Extracting logic circuit description from conjunctive normal form descriptions. In *Proceedings of the IEEE/ACM 20th International Conference on VLSI Design*, pages 37–42. IEEE Computer Society, 2007.

[43] Malay K. Ganai, Pranav Ashar, Aarti Gupta, Lintao Zhang, and Sharad Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proceedings of the 39th Design Automation Conference (DAC'02)*, pages 747–750, 2002.

[44] Malay K. Ganai and Andreas Kuehlmann. On-the-fly compression of logical circuits. In *Informal Proceedings of the 9th IEEE/ACM International Workshop on Logic Synthesis (IWLS'00)*, 2000. Available at `http://citeseer.ist.psu.edu/ganai00fly.html`.

[45] Enrico Giunchiglia, Marco Maratea, and Armando Tacchella. Dependent and independent variables in propositional satisfiability. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors, *Proceedings of the European Conference on Logics in Artificial Intelligence JELIA'02*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 296–307. Springer, 2002.

[46] Enrico Giunchiglia, Alessandro Massarotto, and Roberto Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In B.G. Buchanan C. Rich, J. Mostow and R. Uthurusamy, editors, *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 948–953. AAAI Press, 1998.

[47] Andreas Goerdt. Regular resolution versus unrestricted resolution. *SIAM Journal on Computing*, 22(4):661–683, 1993.

[48] Evgueni Goldberg and Yakov Novikov. Berkmin: A fast and robust SAT-solver. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference (DATE'02)*, pages 142–149. IEEE Computer Society, 2002.

[49] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In B.G. Buchanan C. Rich, J. Mostow and R. Uthurusamy, editors, *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437. AAAI Press, 1998.

[50] ric Gr goire, Richard Ostrowski, Bertrand Mazure, and Lakhdar Sais. Automatic extraction of functional dependencies. In Holger H. Hoos and David G. Mitchell, editors, *Revised Selected Papers of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542 of *Lecture Notes in Computer Science*, pages 122–132. Springer, 2005.

[51] Jan Friso Groote and Hans Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics*, 130(2):157–171, 2003.

[52] Aarti Gupta, Anubhav Gupta, Zijiang Yang, and Pranav Ashar. Dynamic detection and removal of inactive clauses in SAT with application in image computation. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 536–541. ACM, 2001.

[53] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2–3):297–308, 1985.

[54] Marijn Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In Holger H. Hoos and David G. Mitchell, editors, *Revised Selected Papers of the 7th International Conference on Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2005.

[55] Marijn Heule and Hans van Maaren. Aligning CNF- and equivalence-reasoning. In Holger H. Hoos and David G. Mitchell, editors, *Revised Selected Papers of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2005.

[56] Edward A. Hirsch and Arist Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1):91–111, 2005.

[57] John N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[58] Holger H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'02)*, pages 655–660. AAAI Press, 2002.

[59] Holger H. Hoos and Thomas St tzle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000.

[60] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 2318–2323. AAAI Press, 2007.

[61] Jinbo Huang and Adnan Darwiche. A structure-based variable ordering heuristic for SAT. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1167–1172. Morgan Kaufmann, 2003.

[62] Joey Hwang and David G. Mitchell. 2-way vs. d-way branching for CSP. In Peter van Beek, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2005.

[63] Yannet Interian. Backdoor sets for random 3-SAT. In *Informal Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 231–238, 2003. Available at `http://www.cam.cornell.edu/~interian/sat/backdoor.pdf`.

[64] Madhu K. Iyer, Ganapathy Parthasarathy, and Kwang-Ting Cheng. SATORI—a fast sequential SAT engine for circuits. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'03)*, pages 320–325. IEEE Computer Society, 2003.

[65] Paul Jackson and Daniel Sheridan. Clause form conversions for Boolean circuits. In Holger H. Hoos and David G. Mitchell, editors, *Revised Selected Papers of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2005.

[66] Matti J rvisalo. Equivalence checking multiplier designs, 2007. SAT Competition 2007 benchmark description, `http://www.tcs.hut.fi/~mjj/benchmarks/`.

[67] Matti J rvisalo, Tommi Junttila, and Ilkka Niemel . Unrestricted vs restricted cut in a tableau method for Boolean circuits. *Annals of Mathematics and Artificial Intelligence*, 44(4):373–399, 2005.

[68] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In Benjamin K. Kuipers and Bonnie Webber, editors, *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208. AAAI Press, 1997.

[69] Tommi A. Junttila and Ilkka Niemel . Towards an efficient tableau method for boolean circuit satisfiability checking. In John W. Lloyd, Ver nica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Lu s Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proceedings of the 1st International Conference on Computational Logic (CL'00)*, volume 1861 of *Lecture Notes in Computer Science*, pages 553–567. Springer, 2000.

[70] Toni Jussila, Keijo Heljanko, and Ilkka Niemel . BMC via on-the-fly determinization. *International Journal on Software Tools for Technology Transfer*, 7(2):89–101, 2005.

[71] Henry Kautz, David McAllester, and Bart Selman. Exploiting variable dependency in local search. In *Poster Sessions of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, 1997. Available at `http://www.cs.cornell.edu/home/selman/papers-ftp/97.ijcai.dagsat.ps`.

[72] Henry A. Kautz. Deconstructing planning as satisfiability. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'06)*. AAAI Press, 2006.

[73] Henry A. Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363. John Wiley and Sons, 1992.

[74] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI'96)*, pages 1194–1201. AAAI Press, 1996.

[75] Philip Kilby, John Slaney, Sylvie Thi baux, and Toby Walsh. Backbones and backdoors in satisfiability. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'05)*, pages 1368–1373. AAAI Press, 2005.

[76] Jan Kraj ček. *Bounded arithmetic, propositional logic, and complexity theory*, volume 60 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, 1995.

[77] Jan Kraj ček. An exponential lower bound for a constraint propagation proof system based on ordered binary decision diagrams. *Electronic Colloquium on Computational Complexity (ECCC)*, TR07-007, 2007. Available at `http://eccc.hpi-web.de/eccc-reports/2007/TR07-007/index.html`.

[78] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Conference on Design Automation (DAC'03)*, pages 368–371. ACM Press, 2003.

[79] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. Circuit–based Boolean reasoning. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 232–237. ACM, 2001.

[80] Tracy Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, 1992.

[81] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi A. Junttila. Simple bounded LTL model checking. In Alan J. Hu and Andrew K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2004.

[82] Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[83] Chu Min Li. Equivalent literal propagation in Davis-Putnam procedure. *Discrete Applied Mathematics*, 130(2):251–276, 2003.

[84] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In M. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371. Morgan Kaufmann, 1997.

[85] Paolo Liberatore. Complexity results on DPLL and resolution. *ACM Transactions on Computational Logic*, 7(1):84–107, 2006.

[86] Feng Lu, Li-C. Wang, Kwang-Ting Cheng, John Moondanos, and Ziyad Hanna. A signal correlation guided circuit-SAT solver. *Journal of Universal Computer Science*, 10(12):1629–1654, 2004.

[87] In s Lynce and Jo o Marques-Silva. Efficient haplotype inference with boolean satisfiability. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'06)*. AAAI Press, 2006.

[88] Jo o Marques-Silva and Lu s Guerra e Silva. Solving satisfiability in combinational circuits. *IEEE Design & Test of Computers*, 20(4):16–21, 2003.

[89] Jo o P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In Pedro Barahona and Jos J lio Alferes, editors, *Progress in Artificial Intelligence, Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA'99)*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999.

[90] Jo o P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[91] Fabio Massacci. Simplification: A general constraint propagation technique for propositional and modal tableaux. In Harrie C. M. de Swart, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods TABLEAUX'98*, volume 1397 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 1998.

[92] Fabio Massacci. Using Walk-SAT and Rel-sat for cryptographic key search. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 290–295. Morgan Kaufmann, 1999.

[93] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In Benjamin K. Kuipers and Bonnie Webber, editors, *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326. AAAI Press, 1997.

[94] Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In Armin Biere and Carla P. Gomes, editors, *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2006.

[95] David G. Mitchell. Resolution and constraint satisfaction. In Francesca Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 555–569. Springer, 2003.

[96] David G. Mitchell. A SAT solver primer. *Bulletin of the EATCS*, 85:112–132, 2005.

[97] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535. ACM, 2001.

[98] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[99] Sergey I. Nikolenko. Hard satisfiable instances for DPLL-type algorithms. *Journal of Mathematical Sciences*, 126(3):1205–1209, 2005.

[100] Naomi Nishimura, Prabhakar Ragde, and Stefan Szeider. Detecting backdoor sets with respect to Horn and binary clauses. In *Online Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, 2004. Available at `http://www.satisfiability.org/SAT04/programme/51.pdf`.

[101] Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On generating small clause normal forms. In Claude Kirchner and H l ne Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE'98)*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 397–411. Springer, 1998.

[102] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1995.

[103] Duc Nghia Pham, John R. Thornton, and Abdul Sattar. Building structure into local search for SAT. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 2359–2364. AAAI Press, 2007.

[104] David A. Plaisted and Steven A. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:193–304, 1986.

[105] Steven David Prestwich and In s Lynce. Local search for unsatisfiability. In Armin Biere and Carla P. Gomes, editors, *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 283–296. Springer, 2006.

[106] Tuomo Pyh l . Factoring benchmarks for SAT-solvers, 2004. `http://www.tcs.hut.fi/Software/genfacbm/`.

[107] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemel . Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12–13):1031–1080, 2006.

[108] John A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[109] Jan-Willem Roorda and Koen Claessen. SAT-based assistance in abstraction refinement for symbolic trajectory evaluation. In Thomas Ball and Robert B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2006.

[110] Jarrod A. Roy, Igor L. Markov, and Valeria Bertacco. Restoring circuit structure from SAT instances. In *Informal Proceedings of the 2004 International Workshop on Logic Synthesis*, 2004. Available at `http://www.eecs.umich.edu/~imarkov/pubs/misc/iwls04-sat2circ.pdf`.

[111] Yongshao Ruan, Henry A. Kautz, and Eric Horvitz. The backdoor key: A path to understanding problem hardness. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04)*, pages 124–130. AAAI Press, 2004.

[112] Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004. Available at `http://www.cs.sfu.ca/~mitchell/papers/ryan-thesis.ps`.

[113] Sean Safarpour, Aandreas G. Veneris, Rolf Drechsler, and Joanne Lee. Managing don't cares in Boolean satisfiability. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE'04)*, pages 260–265. IEEE Computer Society, 2004.

[114] Roberto Sebastiani. Applying GSAT to non-clausal formulas. *Journal of Artificial Intelligence Research*, 1:309–314, 1994.

[115] Roberto Sebastiani. Lazy satisfiability modulo theories, April 2007. DIT technical report 07-022, submitted for publication. Available at `http://www.dit.unitn.it/~rseba/publist.html`.

[116] Nathan Segerlind. Nearly-exponential size lower bounds for symbolic quantifier elimination algorithms and OBDD-based proofs of unsatisfiability. *Electronic Colloquium on Computational Complexity (ECCC)*, TR07-009, 2007. Available at `http://eccc.hpi-web.de/eccc-reports/2007/TR07-009/index.html`.

[117] Sakari Seitz, Mikko Alava, and Pekka Orponen. Focused local search for random 3-satisfiability. *Journal of Statistical Mechanics: Theory and Experiment*, P06006, 2005.

[118] Sakari Seitz and Pekka Orponen. An efficient local search method for random 3-satisfiability. *Electronic Notes in Discrete Mathematics*, 16:71–79, 2003.

[119] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS, 1996.

[120] Bart Selman and Henry A. Kautz. Domain-independent extension to GSAT: Solving large structured satisfiability problems. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 290–295. Morgan Kaufmann, 1993.

[121] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446. AAAI Press, 1992.

[122] Junhao Shi, G rschwin Fey, Rolf Drechsler, Andreas Glowatz, J rgen Schl ffel, and Friedrich Hapke. Experimental studies on SAT-based test pattern generation for industrial circuits. In *Proceedings of the 6th International Conference on ASIC, volume 2*, pages 967–970. IEEE Computer Society, 2005.

[123] Raymond M. Smullyan. *First-Order Logic*. Dover Publications, 1995.

[124] Zbigniew Stachniak. Going non-clausal. In *Informal Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, 2002. Available at `http://gauss.ececs.uc.edu/Conferences/SAT2002/Abstracts/stachniak.ps`.

[125] Paul Stephan, Robert K. Brayton, and Alberto L.Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, 1996.

[126] Ofer Strichman. Tuning SAT checkers for bounded model checking. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2000.

[127] Stefan Szeider. Backdoor sets for DLL subsolvers. *Journal of Automated Reasoning*, 35(1–3):73–88, 2005.

[128] Paul Tafertshofer and Andreas Ganz. SAT based ATPG using fast justification and propagation in the implication graph. In Jacob K. White and Ellen Sentovich, editors, *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'99)*, pages 139–146. IEEE Computer Society, 1999.

[129] Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving non-clausal formulas with DPLL search. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 663–678. Springer, 2004.

[130] Ashish Tiwari, Carolyn Talcott, Merrill Knapp, Patrick Lincoln, and Keith Laderoute. Analyzing biological pathways using SAT-based approaches. In *Proceedings of Algebraic Biology 2007 Conference*. Springer, 2007. To appear.

[131] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In A.O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*, volume 8 of *Seminars in Mathematics, V.A. Steklov Mathematical Institute, Leningrad*, pages 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, Springer 1983.

[132] Alasdair Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[133] Alasdair Urquhart. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 1(4):425–467, 1995.

[134] Miroslav N. Velev and Randy E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 1999.

[135] Michael Wachter and Rolf Haenni. Propositional DAGs: a new graph-based language for representing boolean functions. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, pages 277–285. AAAI Press, 2006.

[136] Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1173–1178. Morgan Kaufmann, 2003.

[137] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285. ACM, 2001.

ISBN 978-951-22-8907-3
ISSN 1457-7615