# The Effect of Structural Branching on the Efficiency of Clause Learning SAT Solving: An Experimental Study ⋆

Matti Järvisalo⋆⋆ and Ilkka Niemelä

*Helsinki University of Technology (TKK),*
*Department of Information and Computer Science,*
*P.O. Box 5400, FI-02015 TKK, Finland*
`matti.jarvisalo@tkk.fi, ilkka.niemela@tkk.fi`

**Abstract**

The techniques for making decisions (*branching*), play a central role in complete methods for solving structured instances of propositional satisfiability (SAT). Experimental case studies in specific problem domains have shown that in some cases SAT solvers can determine satisfiability faster if branching in the solver is restricted to a subset of the variables at hand. The underlying idea in these approaches is to prune the search space substantially by restricting branching to strong backdoor sets of variables which guarantee completeness of the search. In this paper we present an extensive experimental evaluation of the effects of structure-based branching restrictions on the efficiency of solving structural SAT instances. Previous work is extended in a number of ways. We study state-of-the-art solver techniques, including clause learning and related heuristics. We provide a thorough analysis of the effect of branching restrictions on the inner workings of the solver, going deeper than merely measuring the solution time. Extending previous studies which

⋆ Research supported by Academy of Finland under grants #211025 and #122399. The authors thank Tommi Junttila for several insightful discussions on the topic and for help on understanding the inner workings of the Boolean circuit tools in the BC package [1]. Additionally, the authors thank the anonymous reviewers for several comments and suggestions which helped considerably in improving the article.
⋆⋆Corresponding author. Financial support of HeCSE Graduate School in Computer Science and Engineering, Emil Aaltonen Foundation, Jenny and Antti Wihuri Foundation, the Foundation of Technology TES, and Nokia Foundation is gratefully acknowledged.

have focused on input-restricted branching, we also consider relaxed branching restrictions that are based on underlying structural properties of the variables.

*Key words:* Backdoor sets, branching heuristics, clause learning, problem structure, propositional satisfiability

## 1 Introduction

Propositional satisfiability (SAT) solving procedures (or *SAT solvers*) 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 [2,3], bounded model checking (BMC) of hardware and software [4–7], and electronic design automation applications such as automated test pattern generation (ATPG) [8,9]. Most recently, we have witnessed applications of SAT solving techniques in new exciting fields such as bioinformatics [10,11] and logical cryptanalysis [12,13].

While local search SAT solvers have proven very successful in solving *random* satisfiability problem instances, 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 work. Typical SAT solvers aimed at solving such structured problems are based on the CNF-level (clausal) *Davis–Putnam–Logemann–Loveland* procedure (DPLL) [14,15].

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* (for example, [16–18]), novel *propagation mechanisms* (for example, *binary clause* [19] and *equivalence reasoning* [20,21]), efficient propagator implementations (*watched literals* [18]), *randomization* and *restarts* [22,23], and *clause learning* [24]. Out of these concepts, clause learning can be regarded as the most important progressive step. This is witnessed by a sequence of further improved solvers (see, for example, [24,18,25,26]), and also by theory [27]. 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 non-clause learning solvers incorporate heuristics based on literal counting [16] and/or one-step lookahead [17], branching in clause learning solvers is also driven by learning.

Most clause learning solvers implement variations of—or build on top of—the *variable state independent decaying sum* (VSIDS) heuristic [18] 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 for example in [23], 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, that is, deciding on which variable to next set a value during search, play an important role in the efficiency of search. Due to an increasing need for solving large structural problems, techniques for making effective decisions during search are vital. Intuitively, the inherent structure of the problem domain is reflected in individual variables in the SAT encoding, and making decisions on structurally irrelevant variables may have an exponential effect on the running times of SAT solvers.

In addition to developing more effective (*dynamic*) branching heuristics, a complementary view on branching is provided by the concept of *(static) branching restrictions*. The idea behind branching restrictions is to limit the set of variables the solver is allowed to branch on to a small subset $I$ instead of the set $N$ of all variables in the SAT instance at hand. The solver will then apply its own dynamic heuristics on the variables in $I$. The motivation behind branching restrictions is that, by selecting $I$ so that the solver remains complete, the search space size is radically reduced from the order of $2^{|N|}$ to $2^{|I|}$ where $|I| << |N|$. In this work we focus on studying the effect of such static branching restrictions on the efficiency of state-of-the-art SAT solvers.

An example of a natural branching restriction is provided by the set of so called *input variables*. 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 behavior of the underlying system is dependent on the *input*—initial state, nondeterministic choices, et cetera—of the system. Problems such as ATPG that deal with logical circuit designs serve as additional examples of domains where system input is naturally present. The key point is that since the system behavior is determined by its input, input-restricted branching DPLL remains complete. Furthermore, experimental case studies in specific problem domains [28–30] have shown that in some cases SAT solvers benefit from restricting the variables the solver is allowed to branch on to those variables that model the input of the underlying system.

The concept of a *(strong) backdoor set* [31,32] of variables is closely related to restricting branching so that the resulting solving method is still complete. A *unit propagation backdoor* 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 backdoor set—although not necessarily of *minimal size*—is the set of input variables. A motivation behind the concept of backdoor sets is the intuition of improving search efficiency by *backdoor-based branching*. In other words, the idea is to restrict the SAT solver to branch on a subset of variables which together determine the values of all of the other values through unit propagation. However, deciding whether a backdoor set of a given size exists is intractable in general [33].

With this in mind, knowledge of the underlying structural properties of variables in the instance at hand makes it easier to apply branching restrictions when solving the instance. Unfortunately, the correspondence between structural properties—such as functional dependencies—in a real-world problem and the propositional variables in a CNF encoding of the problem is not evident. However, in SAT based approaches direct CNF encodings of a problem domain are rarely used. Typically, the problem is first encoded as a general propositional formula. *Boolean circuits* (see [34], for example) offer a compact representation for an arbitrary propositional formula $\phi$ in a DAG-like structure which reflects naturally the structure of $\phi$, including functional dependencies. For solving such a general propositional problem encoding, the Boolean circuit is then translated into a equi-satisfiable CNF formula by introducing additional variables for the subformulas of $\phi$. However, the underlying structure is no more evident in the resulting CNF that is fed to the SAT solver. Knowledge of the circuit structure is thus crucial, since without it finding functional dependencies—such as input variables—or other structural properties of the original problem encoding from the resulting CNF formula is nontrivial and requires specialized techniques [35–37].

*In this paper* we present an extensive experimental evaluation of the effect of structure-based branching restrictions on the efficiency of solving structural SAT instances. While clause learning SAT solvers typically work on the CNF level, we derive the branching restrictions from the Boolean circuit structure underlying the CNF formulas. The motivation for the starting point of this work is that the set of input variables—when the underlying circuit structure is known—provides an easily detectable backdoor. Our emphasis is on the interplay between structure-based branching restrictions and typical clause learning based search techniques in modern complete SAT solvers. The aim is to provide a detailed 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. Rather than directly aiming at improving specific SAT solvers through applying branching restrictions, we provide an detailed analysis of the effect of imposing such restrictions on current highly optimized SAT solving technology.

*The novel aspects of this work* include the following. Previous case studies on restricted branching that we are aware of, including [28–30,38], concentrate usually only on running times of solvers, and shed little light on the effect of the restriction to the inner workings of SAT solvers. Furthermore, in many cases modern solver techniques are not used. In contrast, we analyze in detail the effect of input-restricted branching on the effectivity of state-of-the-art clause learning and branching heuristics. Additionally, previous studies consider mainly input-restricted branching as the only structural way of restricting the decision making in SAT solvers. In this work we devise and apply controlled schemes for allowing branching additionally on CNF variables other than inputs based on underlying structural properties of the problems. The structural properties—such as the number of occurrences of subformulas—are determined from the general propositional formulas which are represented as Boolean circuits. We relate the differences in efficiency resulting from different structural properties to the effectivity of clause learning techniques. In addition to extending previous work, this study complements known experimental studies on comparing SAT solver techniques, such as clause learning schemes [39], restarts [23], and comparisons of branching heuristics (see [16,40] for examples).

The rest of the article is structured as follows. In Section 2 we define Boolean circuits, describe the translation from circuits to CNF formulas we use in the experiments in order to apply a state-of-the-art clausal SAT solver in the experiments, and review the main techniques applied in current clause learning SAT solvers. The experiment setup is described in Section 3, and the main results and analysis is presented in Section 4. Before concluding, a review of known results related to branching restrictions is presented in Section 5.

## 2 Background

In this section we review basic concepts related to propositional satisfiability and define constrained Boolean circuits which we use as the representation form for arbitrary propositional 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 applied in this work.

### *2.1 Propositional Satisfiability*

Given a Boolean variable $x$, there are two *literals*, the positive literal, denoted by $x$, and the negative literal, denoted by $\neg x$, where $\neg$ is the negation (not).

A *clause* is a disjunction ($\vee$, or) of distinct literals and a CNF formula is a conjunction ($\wedge$, and) of clauses.

Given a CNF formula $F$, a (partial) *assignment* for $F$ is a (partial) function $\tau : \mathsf{vars}(F) \rightarrow \{\mathbf{t}, \mathbf{f}\}$, where $\mathbf{t}$ and $\mathbf{f}$ stand for *true* and *false*, respectively. With slight abuse of notation, we define for negative literals $\tau(\neg x) = \neg\tau(x)$, where $\neg\mathbf{f} = \mathbf{t}$ and $\neg\mathbf{t} = \mathbf{f}$. A clause is *satisfied* by $\tau$ 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 $\tau$. An assignment $\tau$ *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*

In this work we use *Boolean circuits* (see [34], for example) for representing arbitrary propositional formulas. Boolean circuits 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.

More formally, a Boolean circuit over a finite set $G$ of *gates* is a set $\mathcal{C}$ of equations of the form $g := f(g_1, \ldots, g_n)$, where $g, g_1, \ldots, g_n \in G$ and $f : \{\mathbf{f}, \mathbf{t}\}^n \rightarrow \{\mathbf{f}, \mathbf{t}\}$ 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 $\tau : G \rightarrow \{\mathbf{f}, \mathbf{t}\}$. An assignment $\tau$ 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 $\tau$ 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 $\tau'$ *satisfies* $\mathcal{C}^\tau$ if (i) $\tau'$ 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*.

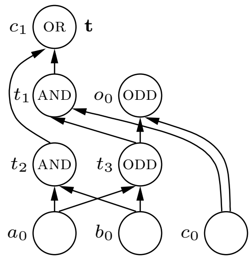
In this work we consider Boolean circuits with the following Boolean functions as gate types:

- $\textsc{not}(v)$ is $\mathbf{t}$ if and only if $v$ is $\mathbf{f}$.
- $\textsc{or}(v_1, \ldots, v_n)$ is $\mathbf{t}$ if and only if at least one of $v_1, \ldots, v_n$ is $\mathbf{t}$.
- $\textsc{and}(v_1, \ldots, v_n)$ is $\mathbf{t}$ if and only if all $v_1, \ldots, v_n$ are $\mathbf{t}$.

- $\text{IMPLY}(v_1, v_2)$ is $\mathbf{t}$ if and only if (i) $v_1$ is $\mathbf{f}$, or (ii) $v_2$ is $\mathbf{t}$.
- $\text{ITE}(v_1, v_2, v_3)$ is $\mathbf{t}$ if and only if (i) $v_1$ and $v_2$ are $\mathbf{t}$, or (ii) $v_1$ is $\mathbf{f}$ and $v_3$ is $\mathbf{t}$.
- $\text{EQUIV}(v_1, \ldots, v_n)$ is $\mathbf{t}$ if and only if (i) all $v_1, \ldots, v_n$ are $\mathbf{f}$, or (ii) all $v_1, \ldots, v_n$ are $\mathbf{t}$.
- $\text{EVEN}(v_1, \ldots, v_n)$ is $\mathbf{t}$ if and only if an even number of $v_1, \ldots, v_n$ are $\mathbf{t}$.
- $\text{ODD}(v_1, \ldots, v_n)$ is $\mathbf{t}$ if and only if an odd number of $v_1, \ldots, v_n$ are $\mathbf{t}$.
- $\text{CARD}_l^u(v_1, \ldots, v_n)$ is $\mathbf{t}$ if and only if at least $l$ and at most $u$ of $v_1, \ldots, v_n$ are $\mathbf{t}$.

**Example 1** *A Boolean circuit and its graphical representation is shown in Figure 1. The circuit models a full-adder with the constraint that the carry-out bit $c_1$ is $\mathbf{t}$. One satisfying truth assignment for the circuit is*

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




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

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

Fig. 1. A constrained Boolean circuit $\langle \mathcal{C}, \tau \rangle$.

### *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 work we apply the CNF translation provided in the BCTools package [1]. The translation procedure works as follows.

(1) The circuit is *normalized* as follows. Non-binary EVEN-, ODD-, and EQUIV-gates are *decomposed*. For example, $g := \text{ODD}(g_1, g_2, g_3)$ is transformed into $g := \text{ODD}(g_1, g')$ and $g' := \text{ODD}(g_2, g_3)$, where $g'$ is a new gate. Based on a heuristic choice, the $\text{CARD}_l^u$-gates are decomposed by (i) applying the equations

$$\text{CARD}_l^u(g_1, \ldots, g_n) \Leftrightarrow \text{CARD}_l^\infty(g_1, \ldots, g_n) \wedge \neg\text{CARD}_{u+1}^\infty(g_1, \ldots, g_n)$$

and

$$\text{CARD}_l^{\infty}(g_1, \ldots, g_n) \Leftrightarrow (g_1 \wedge \text{CARD}_{l-1}^{\infty}(g_2, \ldots, g_n)) \vee \text{CARD}_l^{\infty}(g_2, \ldots, g_n)$$

with dynamic programming, or by (ii) substituting them with a binary adder-comparator circuit when the lower (upper) bound is close to the number of children (close to zero).

(2) The *normalized circuit* $\mathcal{C}^\tau$ resulting from step 1 is translated into CNF with the standard "Tseitin-style" translation. A variable $\tilde{g}$ is introduced for each gate $g$. For encoding the functionalities of gates, gates of the form $g := \text{NOT}(g_1)$ are not translated; instead, $\neg\tilde{g}_1$ is substituted for $\tilde{g}$. For the other gate types, the idea is to represent the logical equivalence $g \Leftrightarrow f(g_1, \ldots, g_n)$ as clauses; hence for each $g := f(g_1, \ldots, g_n)$ the corresponding introduced clauses are as shown in Table 1.

Table 1
CNF translation for constrained Boolean circuits

| gate $g$ | clauses for $g \Rightarrow f(g_1, \ldots g_n)$ | clauses for $f(g_1, \ldots g_n) \Rightarrow g$ |
|---|---|---|
| $g := \text{IMPLY}(g_1, g_2)$ | $(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2)$ | $(\tilde{g} \vee \tilde{g}_1)$, $(\tilde{g} \vee \neg\tilde{g}_2)$ |
| $g := \text{EQUIV}(g_1, g_2)$ | $(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2)$, $(\neg\tilde{g} \vee \tilde{g}_1 \vee \neg\tilde{g}_2)$ | $(\tilde{g} \vee \neg\tilde{g}_1 \vee \neg\tilde{g}_2)$, $(\tilde{g} \vee \tilde{g}_1 \vee \tilde{g}_2)$ |
| $g := \text{EVEN}(g_1, g_2)$ | $(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2)$, $(\neg\tilde{g} \vee \tilde{g}_1 \vee \neg\tilde{g}_2)$ | $(\tilde{g} \vee \neg\tilde{g}_1 \vee \neg\tilde{g}_2)$, $(\tilde{g} \vee \tilde{g}_1 \vee \tilde{g}_2)$ |
| $g := \text{ODD}(g_1, g_2)$ | $(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \neg\tilde{g}_2)$, $(\neg\tilde{g} \vee \tilde{g}_1 \vee \tilde{g}_2)$ | $(\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2)$, $(\tilde{g} \vee \tilde{g}_1 \vee \neg\tilde{g}_2)$ |
| $g := \text{OR}(g_1, \ldots, g_n)$ | $(\neg\tilde{g} \vee \tilde{g}_1 \vee \cdots \vee \tilde{g}_n)$ | $(\tilde{g} \vee \neg\tilde{g}_1)$,...,$(\tilde{g} \vee \neg\tilde{g}_n)$ |
| $g := \text{AND}(g_1, \ldots, g_n)$ | $(\neg\tilde{g} \vee \tilde{g}_1)$,...,$(\neg\tilde{g} \vee \tilde{g}_n)$ | $(\tilde{g} \vee \neg\tilde{g}_1 \vee \cdots \vee \neg\tilde{g}_n)$ |
| $g := \text{ITE}(g_1, g_2, g_3)$ | $(\neg\tilde{g} \vee \neg\tilde{g}_1 \vee \tilde{g}_2)$, $(\neg\tilde{g} \vee \tilde{g}_1 \vee \tilde{g}_3)$ | $(\tilde{g} \vee \neg\tilde{g}_1 \vee \neg\tilde{g}_2)$, $(\tilde{g} \vee \tilde{g}_1 \vee \neg\tilde{g}_3)$ |
| $\langle g, \mathbf{t} \rangle \in \tau$ | $(\tilde{g})$ | |
| $\langle g, \mathbf{f} \rangle \in \tau$ | $(\neg\tilde{g})$ | |

### *2.4 The Anatomy of Modern SAT Solvers*

Most modern complete SAT solvers are based on the DPLL procedure [14,15]. Given a CNF formula $F$ as input, DPLL is a depth-first search procedure building a partial assignment $\tau$ from the variables in $F$ to $\{\mathbf{t}, \mathbf{f}\}$ through (i) *branching* and (ii) *unit propagation* (UP). In branching, the current assignment $\tau$ is extended with $\tau(x) = v$, where $v \in \{\mathbf{f}, \mathbf{t}\}$, for some unassigned variable $x$. Unit propagation extends the current partial assignment $\tau$ with $\tau(l) = \mathbf{t}$ 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 $1 \leq i \leq k$, where $l$ and each $l_i$ are literals. Variables assigned by branching in the current assignment are *decision variables*, and those assigned by UP are *implied variables*.

Most complete SAT solvers aimed at solving structured instances enhance DPLL with *conflict analysis* (or *clause learning*) [24] which is applied when a conflict is reached, that is, when unit propagation would assign the opposite value to an already assigned variable. If there is a conflict at decision level zero, the formula $F$ is determined unsatisfiable. In other cases, the conflict is *analyzed*, and a *learned clause* (or *conflict clause*), which describes the "cause" of the conflict, is added to $F$. After this, clause learning solvers typically apply *non-chronological backtracking* (or *conflict driven backjumping*) based on the conflict clause. We will now give a more detailed intuition into the main techniques centered around clause learning in modern SAT solvers. To complement this description, we refer the reader for example to [24].

A clause is called *known* if it either appears in the original CNF formula or has been learned earlier during the search. Conflict analysis is based on a *conflict graph*, which captures the way the current conflict has been reached. The nodes of the graph are labeled by the variable assignments. There are directed edges from each $\tau(l_i) = \mathbf{f}$ to $\tau(l) = \mathbf{t}$ if and only if the assignment $\tau(l) = \mathbf{t}$ has been made by UP based on the assignments $\tau(l_i) = \mathbf{f}$ with a known clause $(l_1 \vee \cdots \vee l_k \vee l)$. After this, a conflict clause is formed based on a *conflict cut* in the conflict graph.

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 variables currently assigned by branching. A conflict cut is any cut in the conflict graph with all the decision variable assignments on one side (the *reason side*) and at least one of the assignments on the conflict variable 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 assignments, UP can arrive at the conflict at hand. The literals satisfied by the negations of these assignments form the *conflict clause associated with the conflict cut*.

The strategy for fixing a conflict cut is called the *learning scheme*. Typically implemented clause learning schemes are based on *unique implication points* (UIPs) [24]. A UIP in the conflict graph is a node $u$ on the current decision level $d$ such that all paths from the assignment on the decision variable $x$ at level $d$ to the assignments on the conflict variable go through $u$. Such a conflict clause causes the value of the UIP to be immediately flipped by UP when backtracking. UIP learning enables (conflict driven) backjumping in which DPLL non-chronologically backtracks to the maximal decision level of the variables other than the UIP in the conflict clause. A popular version of UIP learning is the 1-UIP scheme, where a cut with the UIP "closest" to the conflict variable assignments is chosen. Different learning schemes are evaluated in [39], showing the robustness of the 1-UIP scheme.

In clause learning solvers, decision heuristics are also typically bound with the clause learning scheme. One popular implementation is the VSIDS heuristic [18] which is based on incrementing the heuristic values of variables/literals associated with conflicts (for example, all literals in the conflict clause [18], or all variables in the conflict clause *and* on the conflict side in each conflict [26]). Furthermore, all heuristic values are decremented by a predetermined factor regularly, typically after every $n$th conflict, with the intuition that the variables causing recent conflicts are especially relevant.

*Restarts* are also 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. Intuitively, restarts help in escaping from getting stuck in hard-to-prove subformulas, and have shown to boost the efficiency of combinatorial search algorithms [22]. An evaluation of the effect of different restart strategies for clause learning SAT solvers is presented in [23].

## 3 Experiment Setup

We evaluate the effect of structural branching restrictions on the behavior of modern clause learning solver techniques. Before detailed discussion of the results, we describe the used Boolean circuit satisfiability benchmarks and BCMinisat, the Boolean circuit front-end for the successful clause learning SAT solver Minisat [26] (version 1.14, available at `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`) which we used for the experiments. BCMinisat is part of the BCTools [1] package developed by Tommi Junttila.

### *3.1 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 was 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 [41], integer factorization based on hardware multiplier designs [42], equivalence checking of hardware multipliers, bounded model checking (BMC) for deadlocks in asynchronous parallel systems modeled as labeled transition systems (LTSs) [43], and linear temporal logic (LTL) BMC of finite state systems with a linear encoding [44].

**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 [41].

**Bounded model checking for deadlocks in LTSs** Circuits resulting 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 [43].

**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) [44].

**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. For a fixed $n$, both multipliers take as input two integers $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$ as binary vectors, and output the product $\mathbf{o} = (o_1, \ldots, o_{2n})$. 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 partial products. The circuits are obtained using the `genfacbm` benchmark generator [42].

**Equivalence checking of hardware multipliers** These circuits encode the problem of equivalence checking the results of the correct adder tree and Braun multipliers. 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 $\mathbf{out} := \textsc{and}(o^{\mathrm{eq}}_1, \ldots, o^{\mathrm{eq}}_{2n})$.
- Constrain **out** to 0 (false).

Since the multiplier designs produce equivalent results for any two multiplicands, we arrive at unsatisfiable equivalence checking instances. In constructing these equivalence checking instances, the braun and atree multiplier circuits were obtained using the `genfacbm` generator [42].

The set of Boolean circuit satisfiability benchmarks (a total of 38 instances) is available at

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

Structural properties of the normalized and simplified circuits are listed in Table 2.

For the experiments, we obtain a total of 570 CNF instances from these circuits as explained next. For solving the instances, 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. The Boolean circuit benchmarks were selected based on the performance of BCMinisat without permuting the CNF variable numbering, so that the running times of BCMinisat on each of the 38 Boolean circuit instances is less than 30 minutes.

### *3.2 Simplification and CNF Translation in BCMinisat*

BCMinisat accepts as input Boolean circuits with the functions listed in Section 2.2 as gate types. The front-end also does circuit-level preprocessing, including Boolean propagation, substructure sharing, and cone–of–influence reduction [45] to the circuit. The resulting normalized and simplified circuit is translated into CNF applying the translation in Section 2.3, and fed to Minisat for solving.

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. Permuting the instances is justified by the fact that the aim of this work is to study the *robustness* of heuristic clause learning SAT solver techniques. This is in contrast with peak performance studies, where the fact that the variable numbering of the CNF instances can reflect the encoded circuit structure may allow for optimizations in the solvers. For example, a solver could be optimized by assuming that input variables are numbered consecutively with small/large numbers.

### *3.3 The Clause Learning CNF Solver Minisat*

Minisat implements 1-UIP clause learning and a variation of the VSIDS heuristic [18]. 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 oc-

Table 2

Properties of the Boolean circuit benchmarks. **sat**: satisfiability of the instance; #inputs: the number of input variables in the CNF translation (percentage out of all variables in parentheses); **circuit heights: max-min/max-max** (maximum of the minimal/maximal length of paths from an output gate to the inputs); **circuit depths: med-min/max-min** (median/maximum of the minimal length of paths from an input gate to the outputs), **med-max** (median of the maximal length of paths from an input gate to the outputs). Notice that here NOT-gates do not contribute to the length of the paths, since they are not translated.

| | | | Height | | Depth | | |
|---|---|---|---|---|---|---|---|
| **Instance** | **sat** | #inputs | max-min | max-max | med-min | max-min | med-max |
| **Super-scalar processor verification** | | | | | | | |
| fvp.2.0.3pipe.1 | no | 186 (8.2) | 5 | 36 | 4 | 5 | 30 |
| fvp.2.0.3pipe_2_ooo.1 | no | 305 (11.7) | 5 | 47 | 1 | 5 | 25 |
| fvp.2.0.4pipe_1_ooo.1 | no | 544 (10.4) | 5 | 75 | 1 | 5 | 27 |
| fvp.2.0.4pipe_2_ooo.1 | no | 547 (9.8) | 5 | 75 | 1 | 7 | 27 |
| fvp.2.0.5pipe_1_ooo.1 | no | 845 (8.9) | 5 | 117 | 1 | 6 | 28 |
| **Equivalence checking hardware multipliers** | | | | | | | |
| eq-test.atree.braun.8 | no | 16 (2.3) | 6 | 30 | 1 | 1 | 30 |
| eq-test.atree.braun.9 | no | 18 (2.0) | 7 | 37 | 1 | 1 | 34 |
| eq-test.atree.braun.10 | no | 20 (1.8) | 7 | 38 | 1 | 1 | 38 |
| **Integer factorization** | | | | | | | |
| atree.sat.34.0 | yes | 60 (0.6) | 11 | 103 | 1 | 1 | 66 |
| atree.sat.36.50 | yes | 64 (0.6) | 10 | 111 | 1 | 1 | 71 |
| atree.sat.38.100 | yes | 68 (0.6) | 11 | 121 | 1 | 1 | 80 |
| atree.unsat.32.0 | no | 57 (0.7) | 10 | 90 | 1 | 1 | 53 |
| atree.unsat.34.50 | no | 60 (0.6) | 11 | 101 | 1 | 1 | 66 |
| atree.unsat.36.100 | no | 64 (0.6) | 10 | 109 | 1 | 1 | 71 |
| braun.sat.32.0 | yes | 61 (2.2) | 3 | 62 | 1 | 1 | 61 |
| braun.sat.34.50 | yes | 65 (2.1) | 3 | 66 | 1 | 1 | 65 |
| braun.sat.36.100 | yes | 69 (2.0) | 3 | 70 | 1 | 1 | 69 |
| braun.unsat.32.0 | no | 60 (2.2) | 3 | 62 | 1 | 1 | 61 |
| braun.unsat.34.50 | no | 64 (2.0) | 3 | 66 | 1 | 1 | 65 |
| braun.unsat.36.100 | no | 68 (1.9) | 3 | 70 | 1 | 1 | 69 |
| **BMC for deadlocks in LTSs** | | | | | | | |
| dp_12.i.k10 | no | 480 (16.0) | 2 | 47 | 1 | 1 | 19 |
| key_4.p.k28 | no | 967 (10.9) | 3 | 56 | 1 | 1 | 20 |
| key_4.p.k37 | yes | 1507 (9.8) | 3 | 74 | 1 | 1 | 26 |
| key_5.p.k29 | no | 1212 (10.7) | 3 | 58 | 1 | 1 | 22 |
| key_5.p.k37 | yes | 1796 (9.8) | 3 | 74 | 1 | 1 | 26 |
| mmgt_4.i.k15 | no | 456 (10.9) | 2 | 33 | 1 | 1 | 19 |
| q_1.i.k18 | no | 566 (13.1) | 2 | 49 | 1 | 1 | 22 |
| **LTL BMC by linear encoding** | | | | | | | |
| 1394-4-3.p1neg.k10 | no | 1845 (5.6) | 4 | 28 | 2 | 4 | 8 |
| 1394-4-3.p1neg.k11 | yes | 2023 (5.5) | 4 | 30 | 2 | 4 | 8 |
| 1394-5-2.p0neg.k13 | no | 1940 (5.0) | 4 | 34 | 2 | 3 | 10 |
| brp.ptimonegnv.k23 | no | 461 (6.7) | 3 | 115 | 2 | 3 | 49 |
| brp.ptimonegnv.k24 | yes | 481 (6.7) | 3 | 120 | 2 | 3 | 51 |
| csmacd.p0.k16 | no | 1794 (2.9) | 3 | 169 | 3 | 13 | 16 |
| dme3.ptimo.k61 | no | 6375 (26.3) | 4 | 603 | 6 | 7 | 470 |
| dme3.ptimo.k62 | yes | 6506 (26.3) | 4 | 615 | 6 | 7 | 480 |
| dme3.ptimonegnv.k58 | no | 5982 (26.3) | 4 | 568 | 6 | 7 | 441 |
| dme3.ptimonegnv.k59 | yes | 6113 (26.3) | 4 | 580 | 6 | 7 | 451 |
| dme5.ptimo.k65 | no | 10750 (26.8) | 4 | 611 | 6 | 7 | 461 |

curred 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 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.

## 4 Results and Analysis

We start by considering the effect of restricting branching in Minisat to inputs variables on the efficiency of the solver. After detailed analysis of input-restricted branching, we will consider the effects of relaxing the input-restriction using various structural properties of the benchmarks. Notice that in the following the main hypotheses are implicitly bound to Minisat since we use Minisat (and its modifications) for the experiments.

### *4.1 Effects of Input-Restricted Branching*

Motivated by the fact that input variables provide an easy-to-detect and relatively small strong backdoor set, and the intuitive drop in the size of the search space achieved by applying input-restricted branching, we consider the following hypothesis as the starting point.

**Hypothesis 1** *The set of input variables, being a relatively small strong backdoor set, provides a branching restriction from which clause learning SAT solvers using the VSIDS heuristic benefit.*

Table 3 gives the minimum, median, and maximum number of decisions for BCMinisat and input-restricted BCMinisat (BCMinisat$_{\mathsf{inputs}}$) for each Boolean circuit satisfiability benchmark instance. For reference, we also include the number of decisions for Minisat (column Minisat) and for Minisat using the SatELite preprocessor (column SatELite) without permuting the CNF variable numbering.

For the instances based on integer factorization and equivalence checking, for which the number of unassigned input variables is 2% or less out of all

Table 3
Number of decisions for Minisat (**Minisat**) and Minisat with the SatELite preprocessor (**SatELite**) without permuting the variable numbering; Minimum (**min**), median (**med**), and maximum (**max**) of number of decisions for BCMinisat and $\text{BCMinisat}_{\mathsf{inputs}}$ with number of timeouts in parenthesis; **ud**: the number of unbranchable variables which have better heuristic values than the best branchable variable per decision for $\text{BCMinisat}_{\mathsf{inputs}}$ (median of averages); **bb**: the fraction of increments on branchable variables from the number of all increments to heuristic values during search (median).

| | Number of decisions | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | BCMinisat | | | $\text{BCMinisat}_{\mathsf{inputs}}$ | | | | |
| **Instance** | **Minisat** | **SatELite** | **min** | **med** | **max** | **min** | **med** | **max** | **ud** | **bb** |
| **Super-scalar processor verification** | | | | | | | | | | |
| fvp.2.0.3pipe.1 | 62449 | 54857 | **61531** | **384386** | **1225134** | - (15) | - (15) | - (15) | - | - |
| fvp.2.0.3pipe_2_ooo.1 | 42183 | 70684 | **75962** | **184798** | **426489** | - (15) | - (15) | - (15) | - | - |
| fvp.2.0.4pipe_1_ooo.1 | 160696 | 204285 | **188992** | **209048** | **271982** | - (15) | - (15) | - (15) | - | - |
| fvp.2.0.4pipe_2_ooo.1 | 305488 | 284037 | **1033607** | **2094617** | **5241781** | - (15) | - (15) | - (15) | - | - |
| fvp.2.0.5pipe_1_ooo.1 | 363318 | 478431 | **336281** | **746231** | **1838599** | - (15) | - (15) | - (15) | - | - |
| **Equivalence checking hardware multipliers** | | | | | | | | | | |
| eq-test.atree.braun.8 | 219092 | 373730 | 180449 | 285665 | 339805 | **65785** | **73834** | **82372** | 88.5 | 0.02 |
| eq-test.atree.braun.9 | 825947 | 1230057 | 898917 | 1055511 | 1317785 | **323688** | **385398** | **389890** | 106.6 | 0.02 |
| eq-test.atree.braun.10 | 4794412 | 6334818 | 3755375 | 4540598 | 5089443 | **1428957** | **1590390** | **1787295** | 127.9 | 0.01 |
| **Integer factorization** | | | | | | | | | | |
| atree.sat.34.0 | 348591 | 142916 | 156733 | 228792 | 761620 | **24820** | **208880** | **277896** | 21.9 | 0.04 |
| atree.sat.36.50 | 495386 | 732282 | **251218** | 721474 | 937152 | 316590 | **571533** | **788762** | 18.4 | 0.04 |
| atree.sat.38.100 | 1152065 | 1057515 | 284980 | 1095192 | - (1) | **190330** | **498092** | **1082729** | - | - |
| atree.unsat.32.0 | 196353 | 174606 | 141419 | 163508 | 180973 | **123502** | **138797** | **162546** | 15.3 | 0.04 |
| atree.unsat.34.50 | 282481 | 290073 | 248371 | 287351 | 404418 | **223130** | **244382** | **301464** | 18.0 | 0.04 |
| atree.unsat.36.100 | 528948 | 816433 | 527237 | 623889 | 915810 | **431576** | **480469** | **578331** | 19.4 | 0.03 |
| braun.sat.32.0 | 96452 | 196844 | 27480 | 82122 | 140150 | **5675** | **81269** | **135093** | 25.6 | 0.05 |
| braun.sat.34.50 | 148494 | 191325 | **30717** | 152224 | 353464 | 43924 | **110614** | **223306** | 25.3 | 0.05 |
| braun.sat.36.100 | 522255 | 615534 | 129771 | 447716 | **589449** | **86134** | **374884** | 752645 | 19.4 | 0.05 |
| braun.unsat.32.0 | 115569 | 157657 | 107617 | 122550 | 156004 | **96894** | **119437** | **150121** | 10.4 | 0.06 |
| braun.unsat.34.50 | 257780 | 311640 | 215624 | 263845 | 341855 | **213199** | **258446** | **316819** | 9.1 | 0.06 |
| braun.unsat.36.100 | 741218 | 635655 | **514725** | **623671** | 807610 | 533575 | 640111 | **674470** | 8.9 | 0.06 |
| **BMC for deadlocks in LTSs** | | | | | | | | | | |
| dp_12.i.k10 | 522359 | 1176291 | **513935** | **639756** | **987595** | 2497570 | - (10) | - (10) | - | - |
| key_4.p.k28 | 102892 | 109422 | **121552** | **147063** | **169386** | 138361 | 184875 | 220107 | 3.7 | 0.53 |
| key_4.p.k37 | 697731 | 286767 | 56784 | **321552** | **1549271** | **7574** | 663152 | - (1) | - | - |
| key_5.p.k29 | 264905 | 216840 | **193139** | **223867** | **310207** | 230844 | 343255 | 405686 | 3.9 | 0.5 |
| key_5.p.k37 | 1069064 | 245622 | 104496 | **421324** | **1540174** | **19027** | 1041807 | - (3) | - | - |
| mmgt_4.i.k15 | 322085 | 332035 | **210288** | **287599** | **457009** | 582998 | 1105986 | 2170048 | 4.2 | 0.41 |
| q_1.i.k18 | 350247 | 282173 | **168156** | **353421** | **507246** | 375493 | 929019 | 1349785 | 3.7 | 0.49 |
| **LTL BMC by linear encoding** | | | | | | | | | | |
| 1394-4-3.p1neg.k10 | 151664 | 105315 | 141822 | 155295 | 164900 | **138468** | **148545** | **156839** | 6.6 | 0.34 |
| 1394-4-3.p1neg.k11 | 236405 | 12190 | 72988 | 128708 | 203647 | **34619** | **55575** | **189434** | 9.0 | 0.32 |
| 1394-5-2.p0neg.k13 | 141788 | 153719 | **125840** | **143928** | **158320** | 146144 | 156527 | 186468 | 6.7 | 0.32 |
| brp.ptimonegnv.k23 | 119438 | 197011 | **106338** | **130577** | **259025** | 193839 | 302930 | 356313 | 4.1 | 0.28 |
| brp.ptimonegnv.k24 | 77692 | 45315 | 43013 | 96775 | **162114** | **13699** | **74907** | 260481 | 5.5 | 0.27 |
| csmacd.p0.k16 | 326348 | 534784 | **229192** | **316082** | **376280** | 269520 | 341751 | 381248 | 4.9 | 0.28 |
| dme3.ptimo.k61 | 438418 | 822795 | **314659** | **549686** | **1658757** | - (15) | - (15) | - (15) | - | - |
| dme3.ptimo.k62 | 815566 | 779476 | **427100** | **688505** | **1545603** | - (15) | - (15) | - (15) | - | - |
| dme3.ptimonegnv.k58 | 700030 | 348887 | **324770** | **568864** | **962967** | - (15) | - (15) | - (15) | - | - |
| dme3.ptimonegnv.k59 | 557504 | 234645 | **303921** | **480073** | **1136938** | - (15) | - (15) | - (15) | - | - |
| dme5.ptimo.k65 | 571986 | 1461903 | **497190** | **735741** | **1839619** | - (15) | - (15) | - (15) | - | - |

unassigned variables, BCMinisat$_{\mathsf{inputs}}$ shows an advantage over BCMinisat with respect to the number of decisions. However, for the hardware verification and BMC instances, the overall performance of BCMinisat$_{\mathsf{inputs}}$ 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.

In Figure 2 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.

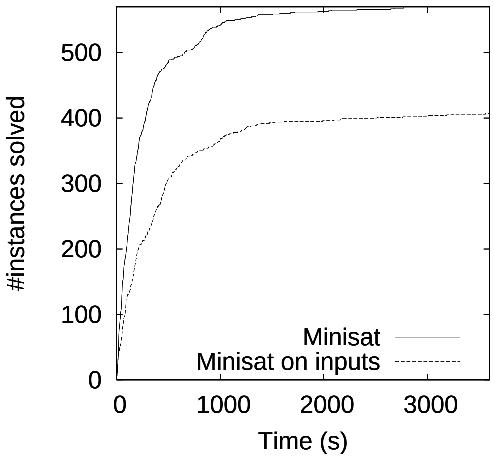



Fig. 2. Comparison of BCMinisat and BCMinisat$_{\mathsf{inputs}}$: cumulative number of solved instances

The effect of input-restricted branching varies depending on whether unsatisfiable or satisfiable instances are considered (Figure 3). 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. However, BCMinisat$_{\mathsf{inputs}}$ does timeout on several satisfiable instances, while BCMinisat timeouts only once.

We make additional observations by looking at statistics over all instances solved by both BCMinisat and BCMinisat$_{\mathsf{inputs}}$.

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*

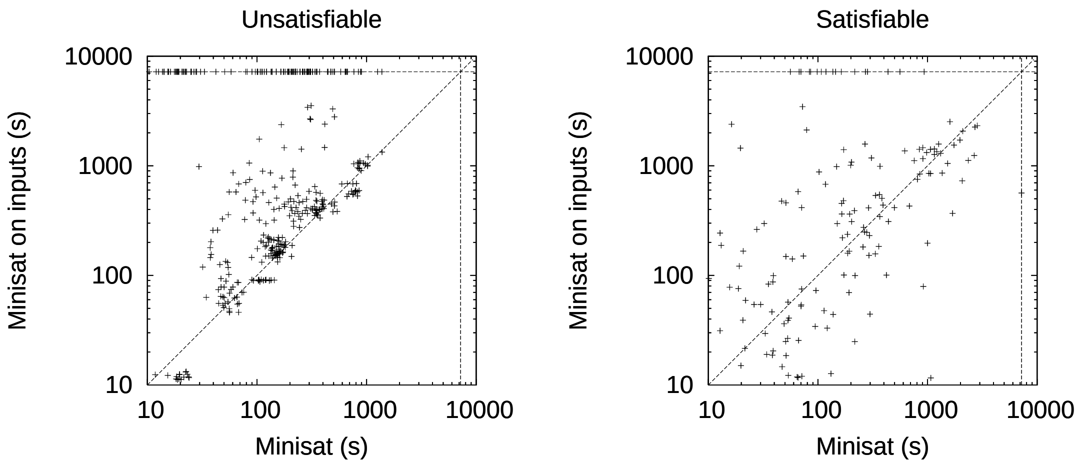



Fig. 3. Comparison of BCMinisat and BCMinisat$_{\mathsf{inputs}}$: running times on unsatisfiable (left) and satisfiable (right) instances

*decisions per second* (see Figure 4; over the set of CNF instances solved by both BCMinisat and BCMinisat$_{\mathsf{inputs}}$).

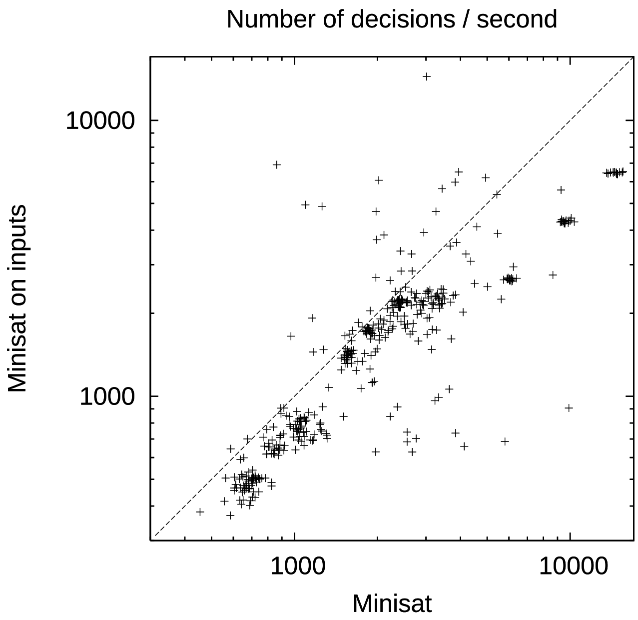



Fig. 4. Comparison of BCMinisat and BCMinisat$_{\mathsf{inputs}}$: number of decisions / second

We also examine the maximal decision levels visited by BCMinisat and BCMinisat$_{\mathsf{inputs}}$ on the different instance families (Figure 5). The intuitive drop in the worst-case behavior of Minisat resulting from input-restricted branching is reflected in the maximal decision levels for the families based on integer factorization and equivalence checking, where the number of input variables is very low (see column #$\mathsf{inputs}$ in Table 2). 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.

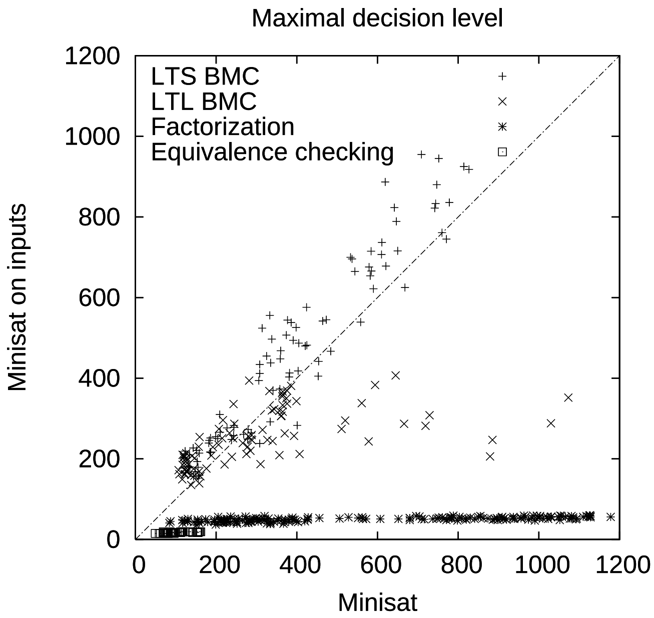



Fig. 5. Comparison of BCMinisat and $\text{BCMinisat}_{\mathsf{inputs}}$: maximal decision levels

On the equivalence checking instances, we notice that the number of decisions for $\text{BCMinisat}_{\mathsf{inputs}}$ *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 $1.4-1.8\times10^6$, compared to the brute-force bound $2^{20} \approx 1.0\times10^6$. Considering that we are using a state-of-the-art clause learning solver, we hypothesize that this surprising result is mostly due to conflict clause forgetting; when forgetting a conflict clause $C$, the solver may have to re-examine the search space characterized as unsatisfiable by $C$. To verify this, we disabled conflict clause forgetting in Minisat (that is, we disabled calls to `reduce_DB()`), and ran this modified Minisat on the `eq-test.atree.braun` benchmarks. Comparison of the minimum, median, and maximum number of decisions for the original $\text{BCMinisat}_{\mathsf{inputs}}$ and the modified $\text{BCMinisat}_{\mathsf{inputs}}$ without conflict clause forgetting is presented in Table 4, along with the search space size when applying input-restricted branching. We notice that with conflict clause forgetting disabled, the number of decisions for $\text{BCMinisat}_{\mathsf{inputs}}$ no more exceeds the search space size. Hence, the conflict clause forgetting mechanism applied in Minisat plays an evident role in causing the peculiar behavior of $\text{BCMinisat}_{\mathsf{inputs}}$ on the equivalence checking instances. However, we note that by disabling conflict clause forgetting, we witnessed an approximately ten-fold increase in the running times of $\text{BCMinisat}_{\mathsf{inputs}}$ on these instances. Conflict clause forgetting seems to be an integral part of the design of Minisat; this also explains why we did not run $\text{BCMinisat}_{\mathsf{inputs}}$ without forgetting on the other instance families.

Table 4
Comparison of the minimum, median, and maximum number of decisions for the original BCMinisat$_{\mathsf{inputs}}$ and BCMinisat$_{\mathsf{inputs}}$ without conflict clause forgetting. The search space size when applying input-restricted branching is given in column $2^{|\mathsf{inputs}|}$.

| | Number of decisions | | | | | | |
|---|---|---|---|---|---|---|---|
| | BCMinisat$_{\mathsf{inputs}}$ | | | BCMinisat$_{\mathsf{inputs}}$ without forgetting | | | |
| Instance | min | med | max | min | med | max | $2^{\lvert\mathsf{inputs}\rvert}$ |
| eq-test.atree.braun.8 | 65785 | 73834 | 82372 | 56286 | 57325 | 57969 | 65536 |
| eq-test.atree.braun.9 | 323688 | 385398 | 389890 | 234077 | 235732 | 238285 | 262114 |
| eq-test.atree.braun.10 | 1428957 | 1590390 | 1787295 | 951504 | 966054 | 975877 | 1048576 |

We have this far observed that, opposed to Hypothesis 1, the clause learning solver Minisat, with the VSIDS heuristics, shows an evident reduction in efficiency when restricting Minisat to branch only on input variables, thus invalidating Hypothesis 1.

However, we have not yet provided explanations for this result, and this will be our next objective. A major aspect is to attempt to give an explanation for the fact that input-restricted branching Minisat manages fewer decision per second.

Since the clause learning mechanism and the VSIDS heuristics, which is tightly bound with the learning mechanism, are key factors in the efficiency of Minisat, we will look for explanations for the performance of BCMinisat$_{\mathsf{inputs}}$ by considering the effect of the input-restriction on the behavior of clause learning and VSIDS. Thus, we consider the following hypotheses.

**Hypothesis 2** *By restricting branching to input variables, clause learning becomes less effective.*

**Hypothesis 3** *By restricting branching to input variables, the solver is forced to make heuristically unimportant decisions.*

An important aspect in the effectiveness of clause learning is the length of conflict clauses, that is, 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 6 we have a comparison of the average lengths of conflict clauses in the solved instances. With input-restricted branching the conflict clauses are typically longer. This supports Hypothesis 2.

Further explanation for the reduced number of decisions per second and the increase in the length of conflict clauses is provided by comparing BCMinisat

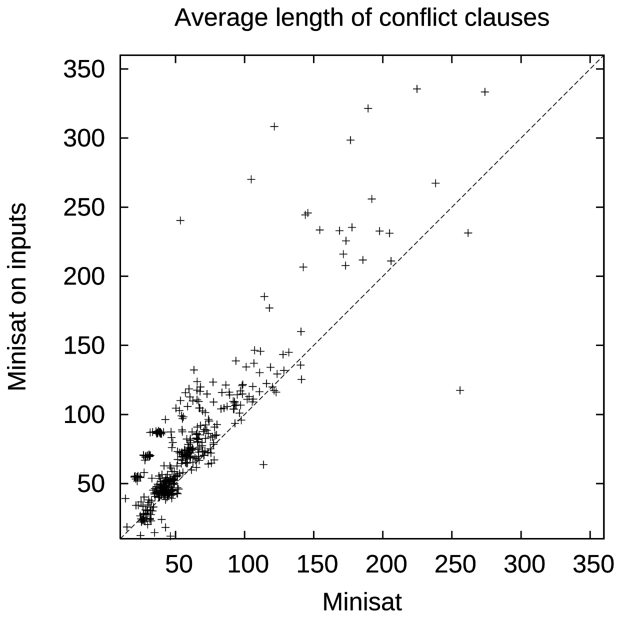



Fig. 6. Comparison of BCMinisat and BCMinisat$_{\mathsf{inputs}}$: average length of conflict clauses

and BCMinisat$_{\mathsf{inputs}}$ with respect to the number of variables assigned by unit propagation (Figure 7) and the number of conflicts per decision (Figure 8).

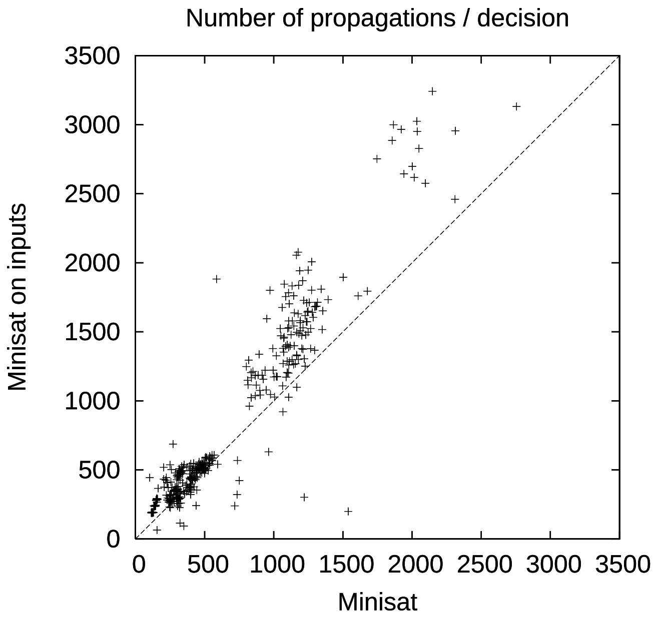



Fig. 7. Comparison of BCMinisat and BCMinisat$_{\mathsf{inputs}}$: number of propagations / decision.

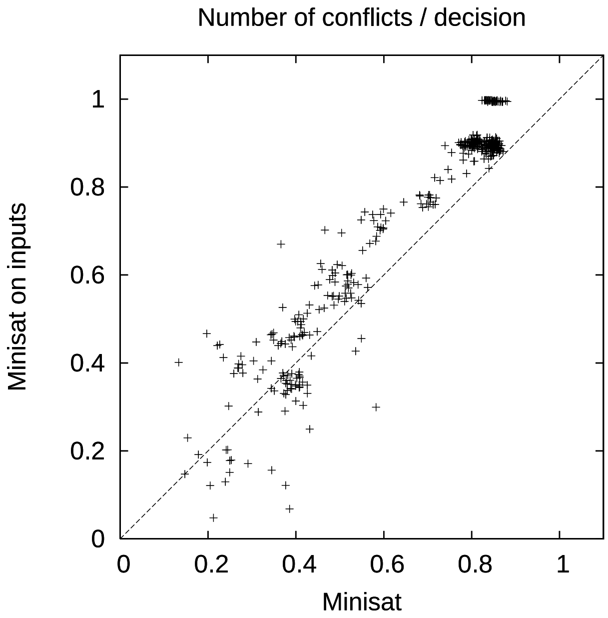



Fig. 8. Comparison of BCMinisat and BCMinisat$_{\mathsf{inputs}}$: number of conflicts / decision.

We observe that, on the average, BCMinisat$_{\mathsf{inputs}}$ does both more propagation per decision and ventures more often into conflicts. At the same time, the conflicts BCMinisat$_{\mathsf{inputs}}$ ventures into result in longer (and thus less effective) conflict clauses using the 1-UIP conflict learning scheme. This leads us to conjecture the following. The combination of increased number of conflicts per decision and propagations per second results in a decrease in the number of decisions the solver is able to make per second. In other words, the input-restricted solver uses relatively more time on propagation and especially, due to the increased number on conflicts per decision, on conflict analysis. Additionally, the increase in time used for conflict analysis does not pay off, since the resulting conflict clauses are longer and thus relatively ineffective. It is very interesting to notice that an increase in the number of propagations does not seem to result in increased solver performance. This is surprising, since it is common to think that the more the solver can unit propagate, the better. It seems that the effectivity of clause learning depends more on the *specific value assignments* that have been made rather than *how many assignments* have been made, and is in support of Hypothesis 3. This is very much in contrast with DPLL solvers without clause learning, in which unit propagation plays an important role in pruning the search space due to standard backtracking and lack of conflict analysis.

Considering Hypothesis 3 further, we observe that the VSIDS heuristic does not seem to 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 3). 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 3). Since the heuristic scores of the variables on which BCMinisat$_{\mathsf{inputs}}$ is allowed to branch are very infrequently updated, the input-restriction results in the risk of degenerating VSIDS into a random heuristic. Together these two observations strongly support Hypothesis 3.

The evidence provided thus far in support of Hypotheses 2 and 3 leads one to question how often the original BCMinisat, without any restriction on which variables to branch on, actually branches on input variables.

**Hypothesis 4** *Input variables are seldom decision variables.*

To investigate the validity of Hypothesis 4, we recorded the number of decisions unrestricted-branching Minisat made on input variables. We calculate for all instances the ratio of the number of decisions Minisat made on input variables ($d_{\mathsf{inputs}}$) to the total number of decisions ($d$) Minisat made on the instance, that is, $\frac{d_{\mathsf{inputs}}}{d}$. We then obtain the *relative number of decisions on input variables* for Minisat by dividing $\frac{d_{\mathsf{inputs}}}{d}$ by the ratio of the number of input variables to the total number of variables in the instance.

The ratio of the running time of the unrestricted-branching Minisat to the running time of input-restricted Minisat as a function of the relative number of decisions on input variables is shown in Figure 9 over all instances on which neither of the solvers timed out. The (i) horizontal and (ii) vertical lines represent the instances on which unrestricted and input-restricted Minisat have identical running times (i) and the instances on which the number of decisions made by unrestricted Minisat on input variables correlates with the percentage of input variables in the instance (ii), respectively The points in the lower left part of the plots represents instances on which the unrestricted-branching Minisat has both branched relatively few times on input variables and solved the instance faster than input-restricted Minisat. If the relative efficiency of unrestricted Minisat would be strongly related with relatively few decision made on input variables, this should show in the plots as most of the points being in either the lower left or the upper right parts. However, such behavior is not visible, as the data points are rather evenly distributed around the horizontal and vertical lines. By this observation it seems that the reason for the difference in running times for unrestricted and input-restricted branching Minisat is not due to unrestricted Minisat making relatively few decisions on input variables, but rather—in disagreement with Hypothesis 4—due to the fact the the unrestricted solver can branch on other relevant variables in addition to inputs.

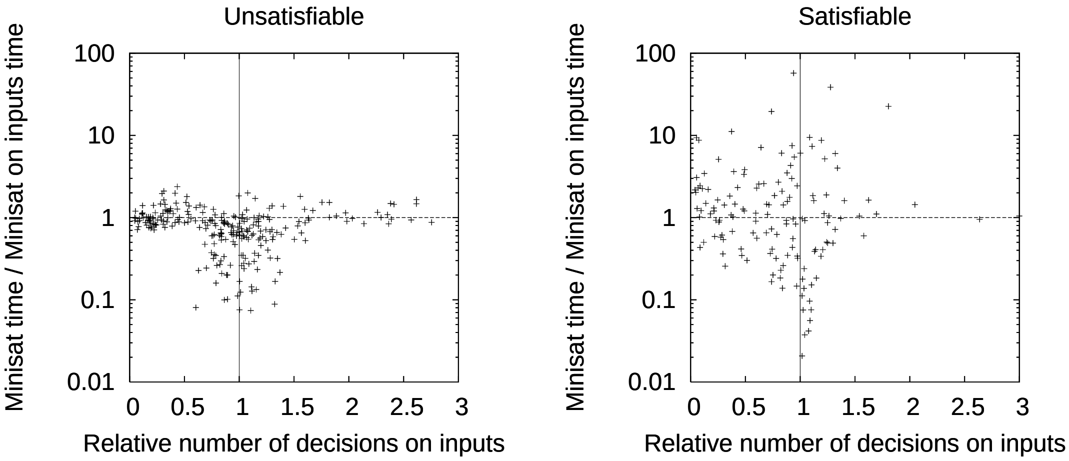



Fig. 9. The ratio of the running time of the unrestricted-branching Minisat to the running time of input-restricted Minisat as a function of the relative number of decisions on input variables: unsatisfiable (left) and satisfiable instances (right)

#### *4.1.1 Conclusions on the Effects of Input-Restricted Branching*

Concerning Hypotheses 1–4, we make the following conclusions based on the experiments on input-restricted branching.

- Although the set of input variables provides a relatively small strong backdoor set, the clause learning SAT solver Minisat, using the VSIDS heuristic, does not benefit from the restriction as such. The performance degrades especially on unsatisfiable instances.
- The effectivity of clause learning degrades. While the input-restricted branching solver ventures into more conflicts per decision, conflict clauses become longer and more time is used on conflict analysis.
- The solver runs into more conflicts and propagates more per decision. However, this does not help in making the search more efficient, since at the same time the conflict clauses become longer and thus less effective.
- The solver is often forced to branch on variables that are unimportant with respect to heuristic scores of VSIDS.

We conjecture that, at least without fundamentally modifying the conflict learning and branching heuristics, it is unlikely that input-restricted branching can be successfully incorporated into clause learning solvers with VSIDS. The evidence against Hypothesis 4 leads us to conjecture that in order to regain robustness of the solver, the input-restriction needs to be relaxed by allowing branching on additional variables.

### 4.2 Effects of Relaxing Input-Restricted Branching

The conclusions on the performance degrading effects of input-restricted branching lead us to the question of how the number of variables on which the solver is allow to branch correlates with solver performance. Can the robustness of input-restricted branching be improved while still branching on a subset of variables? Another aspect is whether structural properties of the variables on which the solver is allowed to branch affect the performance of the solver.

In the following, we 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; that is, we will *relax the input-restriction.*

We will first investigate the following hypothesis.

**Hypothesis 5** *The more relaxed the branching restriction is, the better the restriction works with the solver.*

The first relaxation we consider is the *random restriction*:

**Random restriction (denoted by $\mathsf{rnd}(p)$):** In addition to input variables, branching is allowed on $p$% of randomly chosen unconstrained non-input variables.

Intuitively, this results in allowing branching evenly across the underlying circuit structure. The random restriction will also serve as a reference point for the other structural restrictions we will consider.

We ran BCMinisat with the random restriction with the percentage values $p = 10, 20, 40, 60, 80$. The results as the cumulative number of solved instances, along with input-restricted and unrestricted branching Minisat, are shown in Figure 10. We observe that, inline with Hypothesis 5, allowing branching on non-input variables in addition to inputs, the robustness of the branching-restricted Minisat increases gradually.

For more details on the effects of relaxing the input-restriction, we look at (i) the average length of conflict clauses (on the left in Figure 11) and (ii) the relative number of branchable variables occurring in the conflict clauses (on the right in Figure 11), when moving from input-restricted branching to the random restriction with $p = 20$ (top row in Figure 11), then from the random

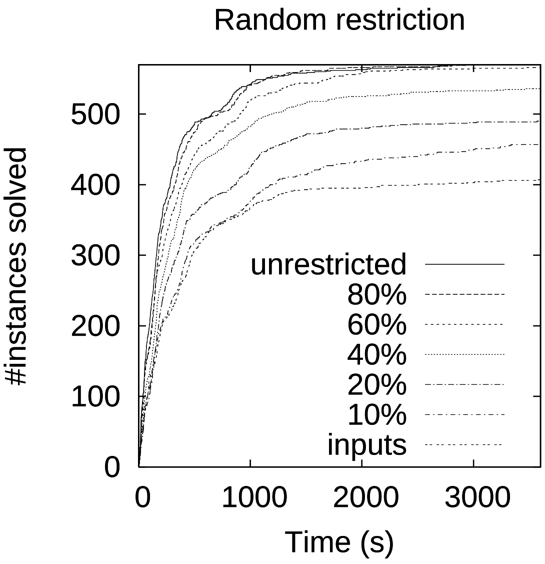



Fig. 10. Cumulative number of solved instances for the random branching restriction

restriction with $p = 20$ to $p = 40$ (middle row in Figure 11), and finally from $p = 40$ to $p = 80$ (bottom row in Figure 11).

We observe that the length of conflict clauses somewhat decreases as $p$ is increased. At the same time, the number of variables on which the solver is allowed to branch evidently increases. For $p = 40$, for example, over half of the literals in the conflict clauses are branchable on most of the instances.

We have now seen that the solver benefits from relaxing the input-restriction. However, the random restriction does not take into account structural properties of the selected variables. Following the intuition behind heuristics found in implementations of DPLL without clause learning—based on literal counting [16] for example—we now turn our attention to the following question. In the context of relaxing the input-restriction, how much do structural properties of the variables on which the solver is additionally allowed to branch affect the relative performance of the solver? Our hypothesis is the following.

**Hypothesis 6** *Structural properties, based on which branching is restricted, play an important role in the efficiency of the solver.*

First, for defining structure-based branching restrictions, we need some additional notation. 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

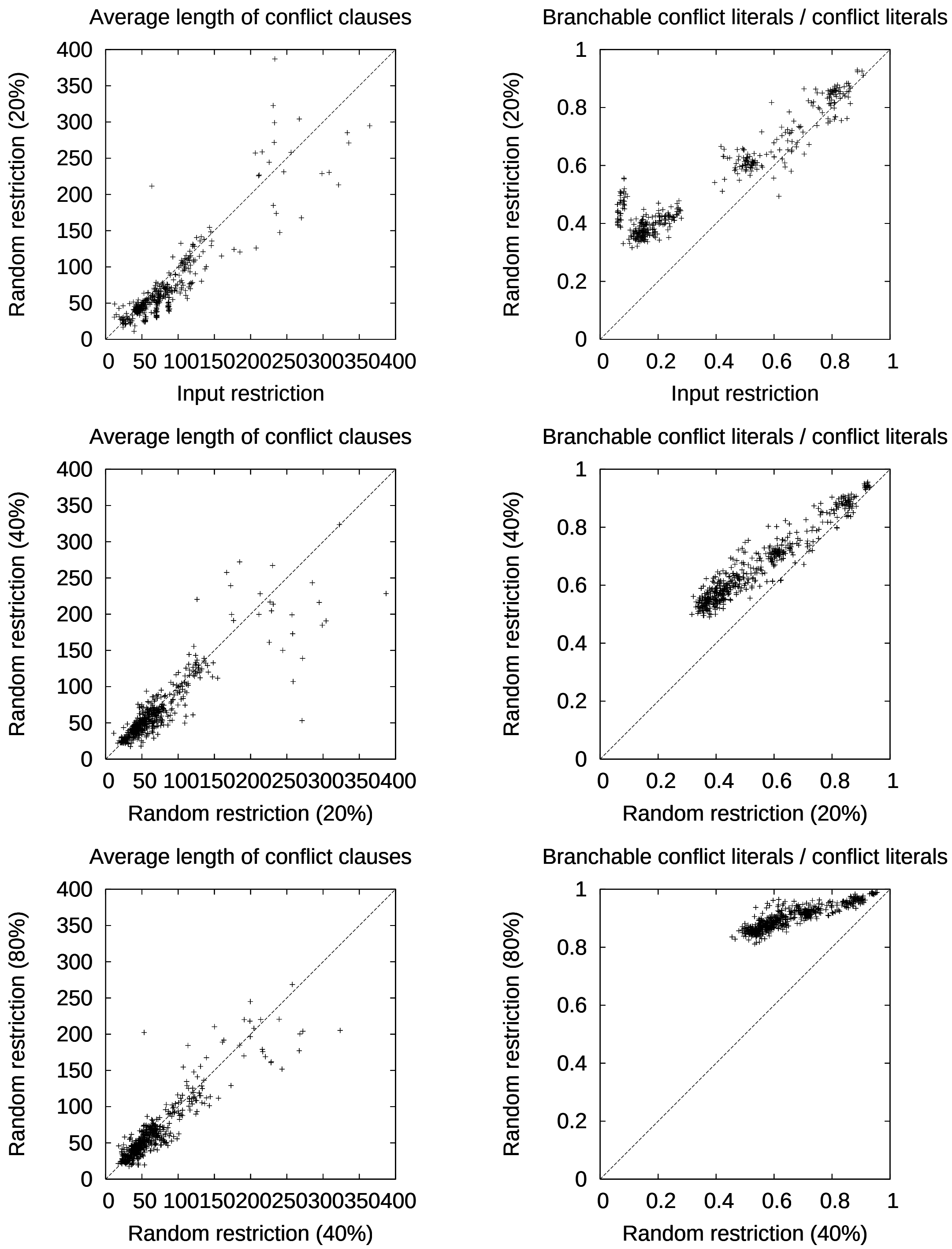



Fig. 11. Comparison of input-restricted branching and random restriction with $p = 20$ (top row), random restriction with $p = 20$ and $p = 40$ (middle row), and random restriction with $p = 40$ and $p = 80$ (bottom row). Left column: average length of conflict clauses, right column: relative number of branchable variables occurring in the conflict clauses.

$g' := \textsc{not}(g)$ 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. Furthermore, $\Delta^{\min}_{\mathsf{inputs}}(g)$ and $\Delta^{\min}_{\mathsf{outputs}}(g)$ denote the lengths of the shortest paths from $g$ to any input and output, respectively.

We are now ready to define the structural branching restriction criteria that we apply.

**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 [38].

**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 [16].

**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 $\tau$. 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 $\tau$, 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 ran BCMinisat with all the above-mentioned branching restrictions and values $p = 10, 20, 40, 60, 80$. The results as the cumulative number of solved instances are shown in Figure 12. 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. The distance-based restrictions result in very poor performance, even compared to the random restriction. In accordance with Hypothesis 6, the choice of the structural criterion does make a difference.

We look for possible explanations for the fact that the structural property based on which branching is restricted affects the efficiency of the solver. We compare the fanout restriction (very close to the original unrestricted solver in performance), the max-min distance restriction (the worst behaving restriction), and also take the random restriction as a reference. The relative number of branchable variables occurring in the conflict clauses and the average length of conflict clauses when using these branching restriction criteria are shown in Figures 13 and 14, respectively, with $p = 20$. As shown in Figure 14, we observe no apparent difference in the lengths of the conflict clauses when comparing the fanout restriction with the max-min distance and random restrictions.

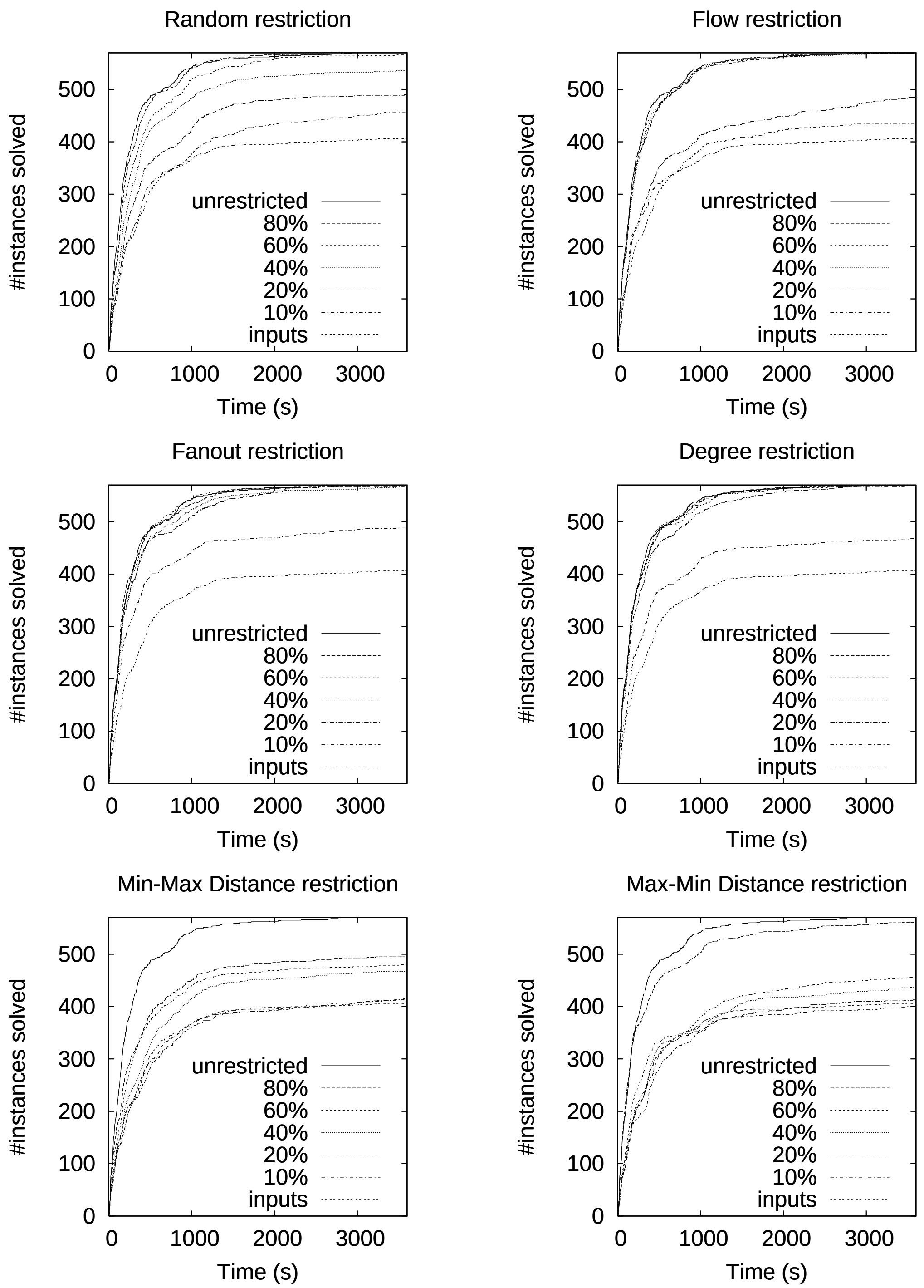



Fig. 12. Cumulative number of solved instances for the structural branching restrictions

However, as shown in Figure 13, we observe a visible difference in the relative number of branchable variables occurring in the conflict clauses. Compared to the other two restrictions, with the fanout restriction the conflict learning mechanism of Minisat produces conflict clauses consisting of a high number of variables on which the solver is allowed to branch.

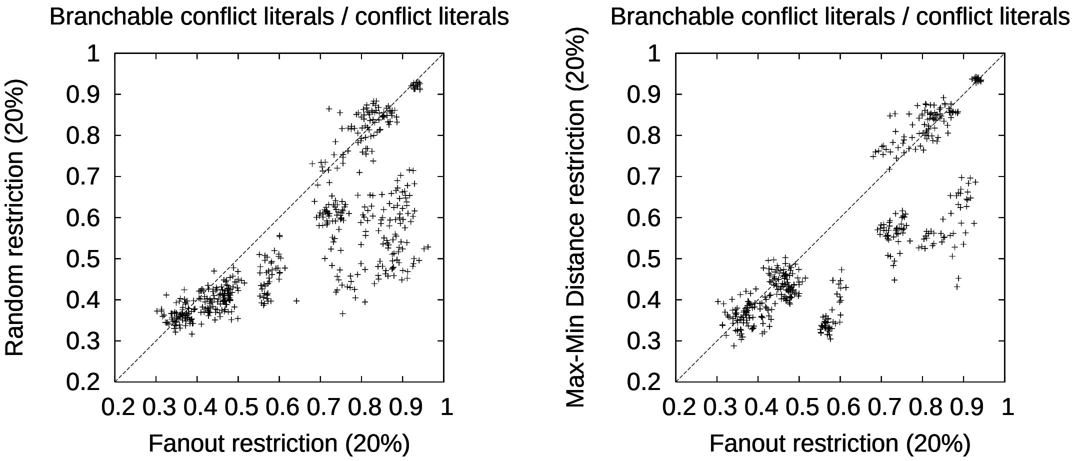



Fig. 13. Comparison of fanout, max-min distance, and random restrictions for $p = 20$: relative number of branchable variables occurring in the conflict clauses

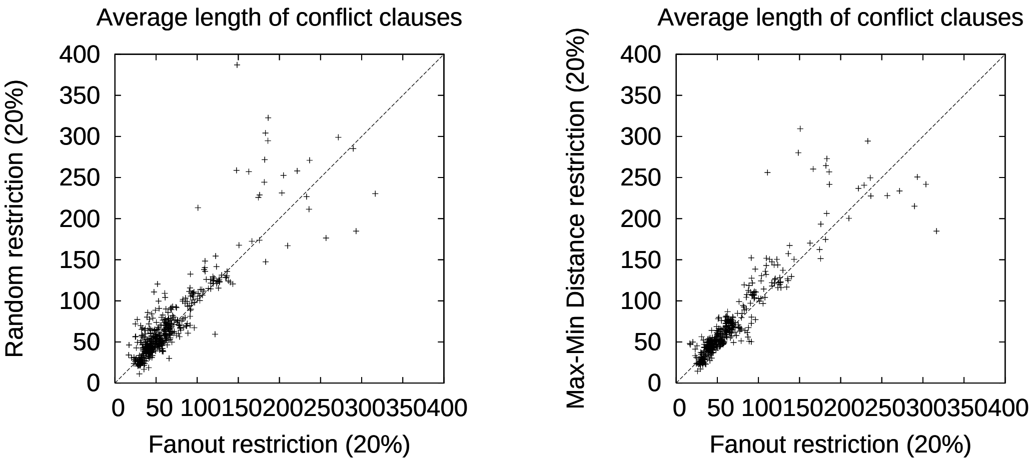



Fig. 14. Comparison of fanout, max-min distance, and random restrictions for $p = 20$: average length of conflict clauses.

This leads us to conjecture the following: the fanout restriction works well with the conflict learning mechanism of the solver because the produced conflict clauses participate actively in the search in the sense that the solver can often branch on variables in the conflict clauses. Thus, we suggest that when restricting branching in a clause learning solver, it is important to ensure that the conflict clauses generated during search contain a high number of variables on which the solver is allowed to branch. A step into this direction is taken in a recent work [46] which studies this possibility in the special case of At-Most-One cardinality constraints.

### *4.2.1 Conclusions on the Effects of Relaxing Input-Restricted Branching*

- Compared to strict branching restrictions, such as the input-restriction, more relaxed branching restrictions allow the solver to better apply its clause

learning and branching heuristics for making search more efficient.
- The choice of the structural criterion based on which branching is restricted plays an important role in the efficiency of the solver; some structural criteria, such as fanout-based, seem to allow rather strict restrictions without loss in efficiency, while other criteria can perform even worse than randomly restricting branching.

We conjecture that the number of variables on which the solver is allowed to branch *in the conflict clauses generated during search* is a determining factor for the efficiency of the branching-restricted solver.

## 5 Related work

### *5.1 Experiments on Branching Restrictions*

In the context of SAT based scheduling, the possibility of restricting branching to inputs (or *control variables*) is suggested in [47], without empirical evaluation, however. For SAT based planning, input-restricted branching (or branching on *action variables*) is studied in [28] showing that the DPLL solver Tableau (having *no clause learning*) benefits from this restriction on a selection of instances. Considering SAT based bounded model checking (BMC), in [29] 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 [30] the authors investigate the effect of restricting branching to inputs (or *independent variables*, calling this the *independent variables set (IVS) heuristic*) on solving 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 [30] is on comparing different decision heuristics that are *not coupled with* clause learning, as opposed to the popular VSIDS heuristic today. Most recently, effect of input-restricted branching on the efficiency of a variety of modern clause learning solvers in the context of SAT based ATPG is studied in [38].

In all of the above-mentioned experimental research the evaluation is based only on the running times of the solvers. In contrast, in this work we relate the performance of a restricted branching solver in-depth to the fundamental techniques in the solver (clause learning, VSIDS heuristics), Moreover, other structural branching restrictions have not been previously studied systematically.

Branching variable orderings for DPLL based on structural information have also been studied [48,49]. 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.

### *5.2 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 [30] it is noted that restricting to independent variables can result in exponential loss of efficiency for DPLL without clause learning. Applying proof complexity theoretic arguments, again considering DPLL without clause learning, [50] studies the effect of input-restriction and, additionally, a variety of other static and dynamic restrictions. 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. Recently, the work in [51] considers the case of input-restricted branching in DPLL *with* clause learning: it is shown that this inference system cannot simulate even the basic DPLL without clause learning.

## 6 Conclusions

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 clause learning based search techniques found in most modern complete SAT solvers. A starting point for this work is provided by the fact that input variables form a relatively small strong backdoor set of variables that is easy detect if a structural representation of the problem in the form of a formula or a circuit is available.

Our novel findings include the following. Although the set of input variables provides a relatively small strong backdoor set, the clause learning SAT solver Minisat, using the VSIDS heuristic, does not benefit from the restriction as such. While the input-restricted branching solver runs into more conflicts per decision, conflict clauses become longer and more time is used on conflict analysis. The solver is often forced to branch on variables that are unimportant with respect to heuristic scores of VSIDS. Compared to input-restricted branching, more relaxed branching restrictions allow the solver to better apply its clause learning and branching heuristics for making search more efficient.

However, the choice of the structural criterion based on which branching is restricted plays an important role in the efficiency of the solver.

Based on the experimental results, we make the following main conjectures.

(i) In order to regain robustness of the solver, the input-restriction needs to be relaxed by allowing branching on additional variables.
(ii) The number of variables on which the solver is allowed to branch *in the conflict clauses generated during search* is a determining factor for the efficiency of the branching-restricted solver.

Conjecture (ii) suggests that, when restricting branching, one way of modifying clause learning with respect to the restriction is to strive to learn clauses with high numbers of variables on which the solver can branch.

A relevant direction of further study is the possibility of restricting branching based on the known structure of known/novel CNF encodings of more general Boolean constraints. A step into this direction is taken in a recent work [46] which studies this possibility in the special case of At-Most-One cardinality constraints. Another interesting direction would be to investigate if solver efficiency could be increased by developing structure-aware branching restriction techniques that act *dynamically* in cooperation with clause learning, especially for Boolean circuit level SAT solvers such as [52]. Furthermore, the experimental analysis on input-restricted branching could be extended to the case of other (possibly minimal) strong backdoor sets.

## References


[1] T. Junttila, The BC package and a file format for constrained Boolean circuits, available at `http://www.tcs.hut.fi/~tjunttil/bcsat/`.

[2] H. A. Kautz, B. Selman, Planning as satisfiability, in: Proc. 10th European Conference on Artificial Intelligence (ECAI 1992), John Wiley and Sons, 1992, pp. 359–363.

[3] J. Rintanen, K. Heljanko, I. Niemelä, Planning as satisfiability: parallel plans and algorithms for plan search, Artificial Intelligence 170 (12–13) (2006) 1031–1080.

[4] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using SAT procedures instead of BDDs, in: Proc. 36th Conference on Design Automation (DAC 1999), ACM Press, 1999, pp. 317–320.

[5] A. Biere, K. Heljanko, T. Junttila, T. Latvala, V. Schuppan, Linear encodings of bounded LTL model checking, Logical Methods in Computer Science 2 (5:5).

[6] D. Kroening, E. Clarke, K. Yorav, Behavioral consistency of C and Verilog programs using bounded model checking, in: Proc. 40th Conference on Design Automation (DAC 2003), ACM Press, 2003, pp. 368–371.

[7] A. Armando, J. Mantovani, L. Platania, Bounded model checking of software using SMT solvers instead of SAT solvers, in: Proc. 13th International SPIN Workshop on Model Checking Software, Vol. 3925 of LNCS, Springer, 2006, pp. 146–162.

[8] T. Larrabee, Test pattern generation using Boolean satisfiability., IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 11 (1) (1992) 4–15.

[9] P. Stephan, R. K. Brayton, A. L. Sangiovanni-Vincentelli, Combinational test generation using satisfiability, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 15 (9) (1996) 1167–1176.

[10] I. Lynce, J. Marques-Silva, Efficient haplotype inference with Boolean satisfiability, in: Proc. 21st National Conference on Artificial Intelligence (AAAI 2006), AAAI Press, 2006.

[11] A. Tiwari, C. Talcott, M. Knapp, P. Lincoln, K. Laderoute, Analyzing biological pathways using SAT-based approaches, in: Proceedings of 2nd International Conference on Algebraic Biology (AB 2007), Vol. 4545 of LNCS, Springer, 2007, pp. 155–169.

[12] I. Mironov, L. Zhang, Applications of SAT solvers to cryptanalysis of hash functions, in: Proc. 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006), Vol. 4121 of LNCS, Springer, 2006, pp. 102–115.

[13] D. De, A. Kumarasubramanian, R. Venkatesan, Inversion attacks on secure hash functions using SAT solvers, in: Proc. 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007), Vol. 4501 of LNCS, Springer, 2007, pp. 377–382.

[14] M. Davis, H. Putnam, A computing procedure for quantification theory, Journal of the ACM 7 (3) (1960) 201–215.

[15] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Communications of the ACM 5 (7) (1962) 394–397.

[16] J. N. Hooker, V. Vinay, Branching rules for satisfiability, Journal of Automated Reasoning 15 (3) (1995) 359–383.

[17] C. M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proc. 15th International Joint Conference on Artificial Intelligence (IJCAI 1997), Morgan Kaufmann, 1997, pp. 366–371.

[18] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in: Proc. 38th Design Automation Conference (DAC 2001), ACM, 2001, pp. 530–535.

[19] F. Bacchus, Enhancing Davis Putnam with extended binary clause reasoning, in: Proc. 19th National Conference on Artificial Intelligence (AAAI 2002), AAAI Press, 2002, pp. 613–619.

[20] C. M. Li, Equivalent literal propagation in Davis-Putnam procedure, Discrete Applied Mathematics 130 (2) (2003) 251–276.

[21] M. Heule, H. van Maaren, Aligning CNF- and equivalence-reasoning, in: Revised Selected Papers of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), Vol. 3542 of LNCS, Springer, 2005, pp. 145–156.

[22] C. P. Gomes, B. Selman, H. A. Kautz, Boosting combinatorial search through randomization, in: Proc. 15th National Conference on Artificial Intelligence (AAAI 1998), AAAI Press, 1998, pp. 431–437.

[23] J. Huang, The effect of restarts on the efficiency of clause learning, in: Proc. 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), AAAI Press, 2007, pp. 2318–2323.

[24] J. P. Marques-Silva, K. A. Sakallah, GRASP: A search algorithm for propositional satisfiability, IEEE Transactions on Computers 48 (5) (1999) 506–521.

[25] E. Goldberg, Y. Novikov, Berkmin: A fast and robust SAT-solver, in: Proc. 2002 Design, Automation and Test in Europe Conference (DATE 2002), IEEE Computer Society, 2002, pp. 142–149.

[26] N. Eén, N. Sörensson, An extensible SAT-solver, in: Revised Selected Papers of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Vol. 2919 of LNCS, Springer, 2004, pp. 502–518.

[27] P. Beame, H. A. Kautz, A. Sabharwal, Towards understanding and harnessing the potential of clause learning, Journal of Artificial Intelligence Research 22 (2004) 319–351.

[28] E. Giunchiglia, A. Massarotto, R. Sebastiani, Act, and the rest will follow: Exploiting determinism in planning as satisfiability, in: Proc. 15th National Conference on Artificial Intelligence (AAAI 1998), AAAI Press, 1998, pp. 948–953.

[29] O. Strichman, Tuning SAT checkers for bounded model checking, in: Proc. 12th International Conference on Computer Aided Verification (CAV 2000), Vol. 1855 of LNCS, Springer, 2000, pp. 480–494.

[30] E. Giunchiglia, M. Maratea, A. Tacchella, Dependent and independent variables in propositional satisfiability, in: Proc. European Conference on Logics in Artificial Intelligence (JELIA 2002), Vol. 2424 of Lecture Notes in Artificial Intelligence, Springer, 2002, pp. 296–307.

[31] R. Williams, C. P. Gomes, B. Selman, Backdoors to typical case complexity, in: Proc. Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003), Morgan Kaufmann, 2003, pp. 1173–1178.

[32] Y. Ruan, H. A. Kautz, E. Horvitz, The backdoor key: A path to understanding problem hardness, in: Proc. 19th National Conference on Artificial Intelligence (AAAI 2004), AAAI Press, 2004, pp. 124–130.

[33] B. Dilkina, C. P. Gomes, A. Sabharwal, Tradeoffs in the complexity of backdoor detection, in: Proc. 13th International Conference on Principles and Practice of Constraint Programming (CP 2007), Vol. 4741 of LNCS, Springer, 2007, pp. 256–270.

[34] C. H. Papadimitriou, Computational Complexity, Addison-Wesley, 1995.

[35] É. Grégoire, R. Ostrowski, B. Mazure, L. Sais, Automatic extraction of functional dependencies, in: Revised Selected Papers of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), Vol. 3542 of LNCS, Springer, 2005, pp. 122–132.

[36] J. A. Roy, I. L. Markov, V. Bertacco, Restoring circuit structure from SAT instances, in: Informal Proc. 2004 International Workshop on Logic Synthesis, 2004, available at `http://www.eecs.umich.edu/~imarkov/pubs/misc/iwls04-sat2circ.pdf`.

[37] Z. Fu, S. Malik, Extracting logic circuit description from conjunctive normal form descriptions, in: Proc. IEEE/ACM 20th International Conference on VLSI Design, IEEE Computer Society, 2007, pp. 37–42.

[38] J. Shi, G. Fey, R. Drechsler, A. Glowatz, J. Schlöffel, F. Hapke, Experimental studies on SAT-based test pattern generation for industrial circuits, in: Proc. 6th International Conference on ASIC, volume 2, IEEE Computer Society, 2005, pp. 967–970.

[39] L. Zhang, C. F. Madigan, M. W. Moskewicz, S. Malik, Efficient conflict driven learning in a Boolean satisfiability solver, in: Proc. 2001 International Conference on Computer-Aided Design (ICCAD 2001), ACM, 2001, pp. 279–285.

[40] J. P. Marques-Silva, The impact of branching heuristics in propositional satisfiability algorithms, in: Proc. 9th Portuguese Conference on Artificial Intelligence (EPIA 1999), Vol. 1695 of LNCS, Springer, 1999, pp. 62–74.

[41] M. N. Velev, R. E. Bryant, Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic, in: Proc. 10th IFIP WG 10.5 Advanced Research Working Conference (CHARME 1999), Vol. 1703 of LNCS, Springer, 1999, pp. 37–53.

[42] T. Pyhälä, Factoring benchmarks for SAT-solvers, `http://www.tcs.hut.fi/Software/genfacbm/` (2004).

[43] T. Jussila, K. Heljanko, I. Niemelä, BMC via on-the-fly determinization, International Journal on Software Tools for Technology Transfer 7 (2) (2005) 89–101.

[44] T. Latvala, A. Biere, K. Heljanko, T. A. Junttila, Simple bounded LTL model checking, in: Proc. 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004), Vol. 3312 of LNCS, Springer, 2004, pp. 186–200.

[45] T. A. Junttila, I. Niemelä, Towards an efficient tableau method for Boolean circuit satisfiability checking, in: Proc. 1st International Conference on Computational Logic (CL 2000), Vol. 1861 of LNCS, Springer, 2000, pp. 553–567.

[46] J. Marques-Silva, I. Lynce, Towards robust CNF encodings of cardinality constraints, in: Proc. 13th International Conference on Principles and Practice of Constraint Programming (CP 2007), Vol. 4741 of LNCS, Springer, 2007, pp. 483–497.

[47] J. M. Crawford, A. B. Baker, Experimental results on the application of satisfiability algorithms to scheduling problems, in: Proc. 11th National Conference on Artificial Intelligence (AAAI 1994), AAAI Press, 1994, pp. 1092–1097.

[48] J. Huang, A. Darwiche, A structure-based variable ordering heuristic for SAT, in: Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), Morgan Kaufmann, 2003, pp. 1167–1172.

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

[50] M. Järvisalo, T. Junttila, I. Niemelä, Unrestricted vs restricted cut in a tableau method for Boolean circuits, Annals of Mathematics and Artificial Intelligence 44 (4) (2005) 373–399.

[51] M. Järvisalo, T. Junttila, Limitations of restricted branching in clause learning, in: Proc. 13th International Conference on Principles and Practice of Constraint Programming (CP 2007), Vol. 4741 of LNCS, Springer, 2007, pp. 348–363.

[52] C. Thiffault, F. Bacchus, T. Walsh, Solving non-clausal formulas with DPLL search, in: Proc. 10th International Conference on Principles and Practice of Constraint Programming (CP 2004), Vol. 3258 of LNCS, Springer, 2004, pp. 663–678.