The Effect of Structural Branching on the Efficiency of Clause Learning SAT Solving

Matti Järvisalo

Laboratory for Theoretical Computer Science
P.O. Box 5400, FI-02015 Helsinki University of Technology (TKK), Finland
matti.jarvisalo@tkk.fi

Abstract. The techniques for making decisions, i.e., *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 benefit from structure-based limitations on which variables the solver is allowed to branch. Mainly, the focus has been on input (or independent) variables. Moreover, existing literature sheds little light on the effect of the restriction to the inner workings of SAT solvers, and in many cases current state-of-the-art solver techniques are not used. In this paper we present an extensive experimental evaluation on the effect of structure-based branching restrictions on the efficiency of solving structural SAT instances. The emphasis is on the interplay of structure-based branching restrictions and clause learning based search techniques found in most modern complete SAT solvers: (i) We investigate the effect of input-branching on the effectiveness of clause learning bound heuristics and conflict clauses. (ii) To study whether the robustness of input-restricted branching can be improved, we apply controlled schemes for allowing branching additionally on CNF variables other than inputs based on structural properties—such as the number of occurrences of sub-formulas—of non-clausal formulas.

1 Introduction

Modern complete SAT solvers provide an efficient way of solving various real-world problems as propositional satisfiability (SAT). Typical SAT solvers aimed at solving such structured problems are based on the CNF-level (clausal) *Davis–Putnam–Logemann–Loveland* procedure (DPLL) [1, 2]. Research on boosting the efficiency of DPLL solvers has concentrated on incorporating techniques such as *intelligent branching heuristics* (e.g., [3–5]), novel *propagation mechanisms* (e.g., *binary clause* [6] and *equivalence reasoning* [7, 8]), efficient propagator implementations (*watched literals* [5]), *randomization* and *restarts* [9, 10], and *clause learning* [11].

Out of these concepts, clause learning can be regarded as the most important progressive step, as witnessed by a sequence of further improved solvers [9, 11, 5, 12, 13], and by theory [14]. 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 [3] and/or one-step lookahead [4, 15, 16], 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 [5], which values the variables that have played an active role in reaching recent conflict. Moreover, clause learning enables *non-chronological backtracking* (or *backjumping*). In fact, as noted e.g. in [17], 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, since irrelevant decisions may have an exponential effect on the running times of SAT solvers, branching schemes play a central role in complete SAT methods especially for solving typically very large real-world problem instances. In addition to developing more effective (*dynamic*) branching heuristics, another complementary view on branching is provided by the concept of *(static) branching restrictions*. In SAT based approaches to structured problems such as bounded model checking [18] (of both hardware and software) and automated planning [19], the CNF encoding is often derived from a transition relation, where the behaviour of the underlying system is dependent on the *input*—initial state, nondeterministic choices, et cetera—of the system. Experimental case studies in specific problem domains [20–22], have shown that, in some cases, SAT solvers benefit from restricting the variables the solver is allowed to branch on to so called *input variables*, corresponding to the input of the underlying system, by letting the solver then apply its own dynamic heuristics to this set of variables. Since the system behaviour is determined by its input, input-restricted branching DPLL remains complete. Intuitively, this changes the worst-case behaviour of DPLL from the order of 2^N to 2^I with $I << N$, where I and N are the number of input variables and all variables in the CNF encoding, respectively.

However, the case studies on restricted branching that we are aware of, including [20–23], consider mainly input-restricted branching as the only structural way of restricting the decision making in SAT solvers, and concentrate usually only on running times of solvers. The existing literature sheds little light on the effect of the restriction to the inner workings of SAT solvers, and, in many cases, current state-of-the-art solver techniques are not used. This is important to notice due to the aforemention fundamental difference between non-clause learning and clause learning solvers.

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

(i) We perform an in-depth investigation into the effect of input-branching on the effectiveness of the clause learning based heuristic VSIDS and conflict clauses.
(ii) In order to study whether the robustness of input-restricted branching can be improved, we devise and apply controlled schemes for allowing branching additionally on CNF variables other than inputs based on structural properties—such as the number of occurrences of sub-formulas—of non-clausal formulas.

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

This study complements known experimental studies on comparing SAT solver techniques, such as clause learning schemes [24], restarts [17], and comparisons of branching heuristics (e.g., [3, 25]). Our aim is to provide a more coherent picture of the effect of branching restrictions on the inner workings of modern clause learning solvers, and to understand how important underlying structural properties of variables are in making decisions in clause learning SAT solvers.

After a more detailed review of known results related to branching restrictions (Sect. 1.1), in Sect. 2 we define Boolean circuits, which we use as the non-clausal representation form for structural SAT problems, and describe the translation from circuits to CNF formulas (Sect. 2.1) we use in the experiments in order to apply to the state-of-the-art clausal SAT solver Minisat [13]. The experiment setup, together with an analysis of the experimental results, are presented in Sect. 4.

1.1 Related work

Experiments on Branching Restrictions. In the context of SAT based scheduling, the possibility of restricting branching to inputs (or *control variables*) is suggested in [26], without empirical evaluation, however. For SAT based planning, input-restricted branching (or branching on *action variables*) is considered in [20], showing that the DPLL solver Tableau (having *no clause learning*) benefits from this restriction on the considered instances. Considering SAT based bounded model checking (BMC), in [21] 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 [22], the authors investigate the effect of restricting branching to inputs (or *independent variables*, calling this the *independent variables set (IVS) heuristic*) on planning, BMC, and crafted SAT instances using the SAT solver Sim. The presented results deal partly with clause learning. However, the emphasis in [22] is on comparing different decision heuristics that are *not coupled with* clause learning, as opposed to the popular VSIDS heuristic today. Most recently, in the context of SAT based automated test pattern generation (ATPG), in [23] the authors investigate the effect of input-restricted branching on the efficiency of a variety of modern clause learning solvers. In addition, the authors also consider *fanout-restricted branching*, in which branching is allowed additionally on variables which are associated with subformulas occurring multiple times in the original non-clausal problem. However, it should be noted that in all of the above-mentioned experimental research, the evaluation is based only on the running times of the solvers; a more in-depth investigation into the real cause of the differences in running times w.r.t. the applied solver techniques is somewhat lacking.

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 [22] it is noted that restricting to independent variables can result in exponential loss of efficiency for DPLL without clause learning. Using the notion of proof complexity, again considering DPLL without clause learning, [27] studies the effect of input-restriction and, additionally, a variety of other static and dynamic restrictions, such as *top–down branching*, which is closely related to the *justification frontier heuristic* (see e.g. [28]) used often in DPLL style Boolean circuit satisfiability solvers applied in electronic design automation (EDA). The result is a relative efficiency hierarchy for the considered restrictions, showing that, for example, input-restricted branching DPLL cannot polynomially simulate top-down branching DPLL, which in turn cannot simulate the standard (unrestricted branching) DPLL. Recently, it has also been show that input–restricted branching DPLL *with clause learning* cannot simulate DPLL [29].

The complexity of making the optimal branching decision during search in DPLL is studied in [30], with the results that, while the problem for the standard DPLL is not on the first level of the polynomial hierarchy ($\Delta_2^{\mathbf{P}}[\log n]$–hard), it may be even harder for restricted-branching DPLL ($\mathbf{NP}^{\mathbf{PP}}$–hard, i.e., spanning the whole polynomial hierarchy, under a certain assumption, see [30]).

Finally, the concept of a *backdoor set* [31] of variables is closely related to restricting branching so that the resulting solving method is still complete. A *unit propagation backdoor set for* DPLL is a set of variables such that, once all of these variables have values, all the other variables are set values by unit propagation. Thus one intuitive backdoor set is the set of input variables. While deciding whether a backdoor set of a given size exists is intractable in general, algorithms for finding small backdoor sets for CNF formulas are developed in, e.g., [32].

2 Boolean Circuits and Propositional Satisfiability

The correspondence between system input of a real-world problem and propositional variables in the flat CNF encoding is not evident. However, in SAT based approaches, direct CNF encodings of a problem domain are rarely used: the problem at hand is typically encoded with a general propositional formula ϕ, which is then translated into a logically equivalent CNF formula by introducing additional variables for the subformulas of ϕ. *Boolean circuits* (see e.g. [33]) 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. The system input of the original problem is also reflected as *input gates* in Boolean circuits.

A Boolean circuit over a finite set G of *gates* is a set $\mathcal{C}$ of equations of form $g := f(g_1, \ldots, g_n)$, where $g, g_1, \ldots, g_n \in G$ and $f : \{\mathbf{f}, \mathbf{t}\}^n \to \{\mathbf{f}, \mathbf{t}\}$ is a Boolean function, with the requirements that (i) each $g \in G$ appears at most once as the left hand side in the equations in $\mathcal{C}$, and (ii) the underlying directed graph $\langle G, E(\mathcal{C}) = \{\langle g', g\rangle \in G \times G \mid g := f(\ldots, g', \ldots) \in \mathcal{C}\}\rangle$ is acyclic. If $\langle g', g\rangle \in E(\mathcal{C})$, then g' is a *child* of g and g is a *parent* of g'. If $g := f(g_1, \ldots, g_n)$ is in $\mathcal{C}$, then g is an f-gate (or of type f), otherwise it is an *input gate*. A gate with no parents is an *output gate*. A (partial) assignment for $\mathcal{C}$ is a (partial) function $\tau : G \to \{\mathbf{f}, \mathbf{t}\}$. An assignment τ is consistent with $\mathcal{C}$ if $\tau(g) = f(\tau(g_1), \ldots, \tau(g_n))$ for each $g := f(g_1, \ldots, g_n)$ in $\mathcal{C}$.

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

Typical Boolean functions for gate types in Boolean circuits are:

- $\mathrm{NOT}(v)$ is **t** iff v is **f**.
- $\mathrm{OR}(v_1, \ldots, v_n)$ is **t** iff at least one of $v_1, \ldots, v_n$ is **t**.
- $\mathrm{AND}(v_1, \ldots, v_n)$ is **t** iff all $v_1, \ldots, v_n$ are **t**.
- $\mathrm{IMPLY}(v_1, v_2)$ is **t** iff (i) v_1 is **f**, or (ii) v_2 is **t**.
- $\mathrm{EQUIV}(v_1, \ldots, v_n)$ is **t** iff (i) all $v_1, \ldots, v_n$ are **f**, or (ii) all $v_1, \ldots, v_n$ are **t**.
- $\mathrm{ITE}(v_1, v_2, v_3)$ is **t** iff (i) v_1 and v_2 are **t**, or (ii) v_1 is **f** and v_3 is **t**.
- $\mathrm{EVEN}(v_1, \ldots, v_n)$ is **t** iff an even number of $v_1, \ldots, v_n$ are **t**.
- $\mathrm{ODD}(v_1, \ldots, v_n)$ is **t** iff an odd number of $v_1, \ldots, v_n$ are **t**.
- $\mathrm{CARD}_l^u(v_1, \ldots, v_n)$ is **t** iff at least l and at most u of $v_1, \ldots, v_n$ are **t**.

These functions are also available in the input language of the Boolean circuit front-end used in the experiments of this paper.

Example. Figure 1 shows a Boolean circuit for a full-adder with the carry-out bit c_1 constrained to **t**. The formal definition of the graphical representation is the constrained circuit $\langle \mathcal{C}, \tau \rangle$, where

$$\mathcal{C} = \{c_1 := \mathrm{OR}(t_1, t_2),\ t_1 := \mathrm{AND}(t_3, c_0),\ o_0 := \mathrm{ODD}(t_3, c_0),\ t_2 := \mathrm{AND}(a_0, b_0),\ t_3 := \mathrm{ODD}(a_0, b_0)\} \text{ and } \tau = \{\langle c_1, \mathbf{t} \rangle\}.$$

One satisfying truth assignment for the circuit is
$\{\langle c_1, \mathbf{t} \rangle, \langle t_1, \mathbf{t} \rangle, \langle o_0, \mathbf{f} \rangle, \langle t_2, \mathbf{f} \rangle, \langle t_3, \mathbf{t} \rangle, \langle a_0, \mathbf{t} \rangle, \langle b_0, \mathbf{f} \rangle, \langle c_0, \mathbf{t} \rangle\}$.

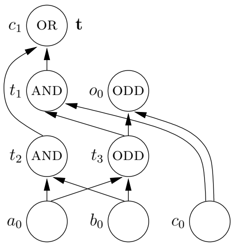

Fig. 1: A constrained circuit.

2.1 Translating Boolean Circuits to CNF

In order to exploit clausal SAT solvers in solving instances of Boolean circuit satisfiability, the circuits have to be translated to CNF. In this paper we apply the following variation of the standard "Tseitin-style" translation. First, the circuit is normalized as follows. Non-binary EVEN-, ODD-, and EQUIV-gates are removed by rewriting them, e.g., $g := \mathrm{ODD}(g_1, g_2, g_3)$ is transformed into $g := \mathrm{ODD}(g_1, g')$ and $g' := \mathrm{ODD}(g_2, g_3)$, where g' is a new gate. Based on a heuristic choice, the CARD_l^u-gates are rewritten by (i) applying the equations $\mathrm{CARD}_l^u(g_1, \ldots, g_n) \Leftrightarrow \mathrm{CARD}_l^\infty(g_1, \ldots, g_n) \wedge \neg\mathrm{CARD}_{u+1}^\infty(g_1, \ldots, g_n)$ and $\mathrm{CARD}_l^\infty(g_1, \ldots, g_n) \Leftrightarrow (g_1 \wedge \mathrm{CARD}_{l-1}^\infty(g_2, \ldots, g_n)) \vee \mathrm{CARD}_l^\infty(g_2, \ldots, g_n)$ with dynamic programming, or (ii) substituting them with a binary adder-comparator circuits when the lower (upper) bound is close to the number of children (close to zero). Finally, the CNF translation of the resulting constrained circuit $\langle \mathcal{C}, \tau \rangle$ is achieved by introducing a variable $\tilde{g}$ for each each gate g. For encoding the functionalities of gates, gates of form $g := \mathrm{NOT}(g_1)$ 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 a normalized Boolean circuit

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

3 The Anatomy of Modern SAT Solvers

Most modern complete SAT solvers are based on the DPLL procedure [1, 2]. Given a CNF formula F as input, DPLL is a depth-first search procedure building a partial assignment τ from the variables in F to $\{\mathbf{t}, \mathbf{f}\}$ through (i) *branching* and (ii) *unit propagation* (UP). In branching, the current assignment τ is extended with $\tau(x) = v$, $v \in \{\mathbf{f}, \mathbf{t}\}$, for some unassigned variable x. Variables assigned by branching in the current assignment are *decision variables*, and those assigned by UP are *implied variables*. Unit propagation extends the current partial assignment τ 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. Here we abuse notation a bit, and define for negative literals $\tau(\neg x) = \neg\tau(x)$, where $\neg\mathbf{f} = \mathbf{t}$ and $\neg\mathbf{t} = \mathbf{f}$. An assignment is extended until (i) some variable x is assigned both $\mathbf{f}$ and $\mathbf{t}$ (a *conflict* is reached, with x as the *conflict variable*) or (ii) τ satisfies F and DPLL terminates. In case (i), non-clause learning DPLL solvers *backtrack* to the last branching decision whose other branch has not been tried yet, undoing all assignments made by UP after the particular decision, and flip the decision. However, most complete SAT solvers aimed at solving structured instances enhance DPLL with *conflict analysis* (or *clause learning*) [11] which is applied when a conflict is reached. 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. However, we refer the reader to, e.g.,[11] for a more concise description of clause learning.

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 an *implication graph*, which captures the way the current conflict has been reached. The nodes of the graph are labeled by the assignments. There are directed edges from each $\tau(l_i) = \mathbf{f}$ to $\tau(l) = \mathbf{t}$ iff 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 implication 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 implication 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) [11]. A UIP in the implication 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 u always exists, since x satisfies this condition; intuitively u is a *single* reason for the conflict at level d. Thus one can always choose a conflict cut that results in a conflict clause with a UIP as the only variable from the maximal decision level. Such a conflict clause causes the value of the UIP to be immediately flipped by UP when backtracking. Furthermore, 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 [24], 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 [5], which is based on incrementing the heuristic values of variables/literals associated with the conflict (e.g., all literals in the conflict clause [5], or all variables in the conflict clause *and* on the conflict side in each conflict [13]). Furthermore, all heuristic values are decremented by a predetermined factor regularly, typically after every nth 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 [9, 10]. A recent evaluation of the effect of different restart strategies for clause learning SAT solvers is [17].

4 Experiments

We now describe our experiments on structural branching restrictions. Before detailed discussion of the results, we describe the used Boolean circuit satisfiability benchmarks and the Boolean circuit front-end BCMinisat applied in solving the instances.

4.1 Benchmarks

Verification of superscalar processors These Boolean circuits encode the problem of formally verifying the correctness of pipelined superscalar processors. The circuits result from a translation from the logic of equality with uninterpreted functions to propositional logic [34].

Integer factorization based on hardware multiplier designs These circuits encode the problem of finding factors for (both divisible and prime) numbers. The problem encodings are based on two structurally very unsimilar hardware binary multiplier designs, the *adder tree* and *Braun* multipliers. The circuits are obtained using the `genfacbm` benchmark generator [35].

Equivalence checking of hardware multipliers These circuits encode the problem of equivalence checking the results of the correct adder tree and Braun multipliers, as described in [36].

Bounded model checking for deadlocks in LTSs These circuits result from a translation scheme (using so called *interleaving* and *process semantics*) for expressing bounded model checking (BMC) of deadlocks in a variety of asynchronous systems modeled as labeled transition systems (LTSs) [37].

Linear temporal logic BMC of finite state systems Linear size Boolean circuit encodings of the BMC problem for finding bugs in finite state system designs violating properties specified in linear temporal logic (LTL) [38].

The set of Boolean circuit satisfiability benchmarks (a total of 38 instances, as detailed in Table 2) is available at `http://www.tcs.hut.fi/~mjj/benchmarks/`. For the experiments, we obtain a total of 570 CNF instances from these circuits as explained next.

4.2 Solving the Instances

For solving the Boolean circuit instances, we apply BCMinisat[1] (version 0.26). BCMinisat is a Boolean circuit front-end for the successful clause learning SAT solver Minisat [13] (version 1.14). BCMinisat accepts as input Boolean circuits with various Boolean functions allowed as gate types, performs circuit-level preprocessing, including Boolean propagation, substructure sharing, and cone-of-influence reductions to the circuit, normalizing the circuit into a form which can be translated into CNF applying the translation detailed in Sect. 2.1. Notice that, when considering structural properties of variables in the resulting CNF formula, the properties are determined by the simplified and normalized circuit, in which gates reflect one-to-one with the CNF variables. For example, an *input variable* is a variable that corresponds to an input gate in the simplified and normalized circuit, and we will take the liberty of using the terms "gate" and "variable" synonymously. BCMinisat feeds the resulting CNF translations and the input-restriction to Minisat, which then solves the CNF. For each circuit, 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 instances 570.

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

Minisat implements 1-UIP clause learning and a variation of the VSIDS heuristic. 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 is decremented by 5%. To avoid hindering efficiency by learning massive amounts of clauses, the solver also uses a scheme for forgetting learned clauses that have not occurred on the conflict side in recent conflicts. Additionally, restarts are used.

We have implemented the considered structural branching restrictions to BCMinisat, and modified Minisat so that its branching and heuristic can be restricted to a given set of variables. We use PCs with 2-GHz AMD 3200+ processors and 2 GBs of memory running Debian GNU Linux, with a 1-hour timeout and a 1-GB memory limit.

4.3 Experiment 1: Effects of Input-Restriction

Table 2 gives the minimum, median, and maximum number of decisions for BCMinisat and input-restricted BCMinisat (BCMinisat$_{\mathsf{inputs}}$) for each benchmark instance. For the instances based on hardware multiplier designs, for which the number of unassigned input variables is 2% or less out of all unassigned variables, BCMinisat$_{\mathsf{inputs}}$ shows an advantage over BCMinisat w.r.t. 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-restriction seems to correlate with a very low relative number of input variables. On the equivalence checking instances, the number of decision for BCMinisat$_{\mathsf{inputs}}$ *is more than the brute-force upper bound*, e.g., for `eq-test.atree.braun.10` around $1.4 - 1.8 \times 10^6$, compared to the brute-force bound $2^{20} \approx 1.0 \times 10^6$. Considering that we are using a state-of-the-art clause learning solver, this surprising result is most likely due to conflict clause forgetting; when forgetting a conflict clause C, the solver may have to re-examine the search space characterised as unsatisfiable by C.

In Fig. 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-restriction. The effect of input-restriction varies depending on whether unsatisfiable or satisfiable instances are considered (Fig. 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. A noticeable point is that, while BCMinisat$_{\mathsf{inputs}}$ makes less decisions for example on the equivalence checking instances, unrestricted BCMinisat is at least as efficient as BCMinisat$_{\mathsf{inputs}}$ when looking at running times. Interestingly, this is due to the fact that unrestricted BCMinisat often *manages more decisions per second* (left in Fig. 4 over all instances solved by both BCMinisat and BCMinisat$_{\mathsf{inputs}}$).

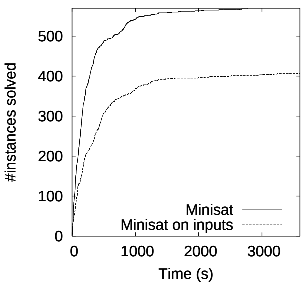

Fig. 2: Solved instances

Table 2. Minimum (**min**), median (**med**), and maximum (**max**) of number of decisions for BCMinisat and BCMinisat$_{\mathsf{inputs}}$, with number of timeouts in parenthesis. The **sat** column gives the satisfiability of the instance, and #inputs gives the number of unassigned input variables in the CNF translation (percentage in parentheses). For **ud** and **bb**, see the text body.

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

We can make more interesting observations by looking at statistics over all instances solved by both BCMinisat and BCMinisat$_{\mathsf{inputs}}$. An important aspect in the effectiveness of clause learning are the lengths of learned clauses, i.e., the number of literals in the clauses. Since a conflict clause describes an unsatisfiable part of the search space, shorter conflict clauses are intuitively exponentially more effective than longer ones. On right in Fig. 4 we have a comparison of the average lengths of learned clauses in the

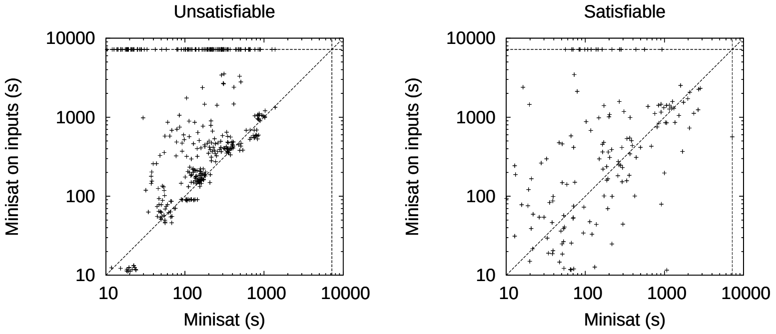

Fig. 3. Scatter plots: running times on unsatisfiable (left) and satisfiable (right) instances

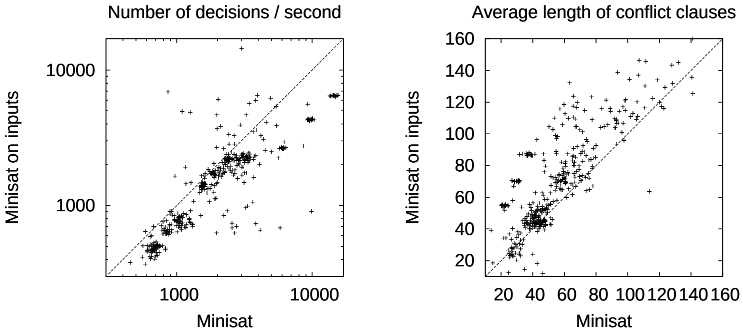

Fig. 4. Scatter plots: number of decisions / second (left), average length of conflict clauses (right)

solved instances. With the input-restriction the learned clause are typically evidently longer. Longer learned clauses can also affect negatively to the efficiency of the solver, since handling the clauses can take more time, e.g., to propagate. This would partly explain the decrease in the number of decisions per time unit on the input-restriction.

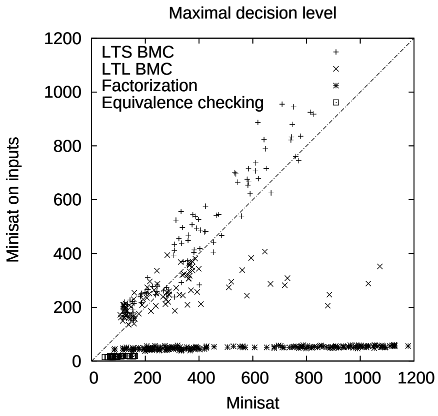

Fig. 5: Maximal decision levels

We also look at the maximal decision levels visited by BCMinisat and $\text{BCMinisat}_{\mathsf{inputs}}$ on the different instance families (Fig. 5). The intuitive drop in the worst-case behaviour of DPLL resulting from the input-restriction is reflected in the maximal decision levels for the families based on multiplier designs, where the number of input variables is very low (see #inputs in Table 2). For the LTS BMC instances, however, the decision levels are greater for the input-restricted solver, although the number of input variables is still only around 10% out of all unconstrained variables.

We also observe that the VSIDS heuristic might not work as intended with the input-restriction. The number of unbranchable variables which have better heuristic values than the best branchable variable can be high per decision (median of averages: **ud** in Table 2), e.g., for `eq-test.atree.braun.10` on the average there are, per decision, over 100 unbranchable variables with better heuristic scores than the best branchable one. From another point of view, the fraction of increments on branchable variables from the number of all increments to heuristic values during search can be in some cases even as low as 1% (median: **bb** in Table 2)—running the risk of VSIDS degenerating into a random heuristic.

These observations imply that in order to incorporate branching restrictions in clause learning solvers, the restriction itself should be taken into account in developing suitable heuristics and learning schemes.

4.4 Experiment 2: Systematic Structural Branching Restrictions

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

For the following, let $\mathcal{C}^\tau$ be a simplified and normalized constrained circuit with the sets of unconstrained gates G, input gates $\mathsf{inputs}(\mathcal{C}^\tau)$, and output gates $\mathsf{outputs}(\mathcal{C}^\tau)$. For a gate $g := f(g_1, \ldots, g_n)$, the set of g's children is $\mathsf{children}(g) = \{g_1, \ldots, g_n\}$, and the set of g's parents is $\mathsf{parents}(g)$. For a gate $g \in G$, the *fanout* $\mathsf{fanout}(g)$ is the number of gates whose child g or $g' := \mathrm{NOT}(g)$ 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.

We will investigate the following criteria.

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

Fanout-based restriction $\mathsf{fan}(p)$**:** 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 [23].

Degree-based restriction $\mathsf{deg}(p)$**:** 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 [3].

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

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

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

Distance-based restriction $\mathsf{dist}(p)$**:** Complementing the other restrictions based on the underlying DAG-structure of Boolean circuits, we also consider restricting branching based on the distances of gates from inputs to outputs:

- In $\mathsf{minmax} - \mathsf{dist}(p)$ gates are ranked according to $\max\{\Delta^{\max}_{\mathsf{inputs}}(g), \Delta^{\max}_{\mathsf{outputs}}(g)\}$, with the criterion that gates with small values are preferred. 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 $\min\{\Delta^{\min}_{\mathsf{inputs}}(g), \Delta^{\min}_{\mathsf{outputs}}(g)\}$, with the criterion that gates with large values are preferred. Branching is concentrated on variables that are far from both input and output variables.

In selecting the best $p\%$ of variables according to a particular criterion, ties are broken randomly from the set of variables having the *break value* of the criterion. For example, consider $\mathsf{fan}(p)$. Let k be the break value such that $100 \times |\{g \mid \mathsf{fanout}(g) \geq k\}|/|G| \geq p$ and $100 \times |\{g \mid \mathsf{fanout}(g) \geq k+1\}|/|G| < p$ hold. Now branching is allowed on all gates g with $\mathsf{fanout}(g) \geq k+1$ and additionally on a number of randomly chosen gates g with the break value $\mathsf{fanout}(g) = k$ so that the percentage p is reached.

We run BCMinisat with all the above-mentioned branching restrictions and values $p = 10, 20, 40, 60, 80$. The results as the cumulative number of solved instance are shown in Fig. 6. First, as witnessed by the random restriction, by allowing branching additionally on non-input variables the robustness of Minisat increases gradually. Considering the structural restrictions, it is interesting to see that for the fanout and degree based restrictions only 20% additional branching variables are enough for the restrictions to reach a level of robustness very close to unrestricted branching Minisat. For the flow-based restriction, this holds from 40% on. It is very interesting to see that the choice of the structural criterion *does make a difference*: we observe that the distance-based restrictions result in very poor performance. In fact, the only restrictions on which Minisat solves all the CNF instances are $\mathsf{deg}(20)$, $\mathsf{deg}(40)$, and $\mathsf{flow}(40)$ (e.g., unrestricted Minisat time outs on one instance out of the 570 CNFs, see Table 2). From these results we draw the conclusion that branching *can be restricted* even rather heavily without losing much of the robustness of a clause learning SAT solver on various instance families. On the other hand, at least the considered *structural restrictions do not seem to be beneficial in general* on their own, since none of them give notable gains w.r.t. unrestricted branching.

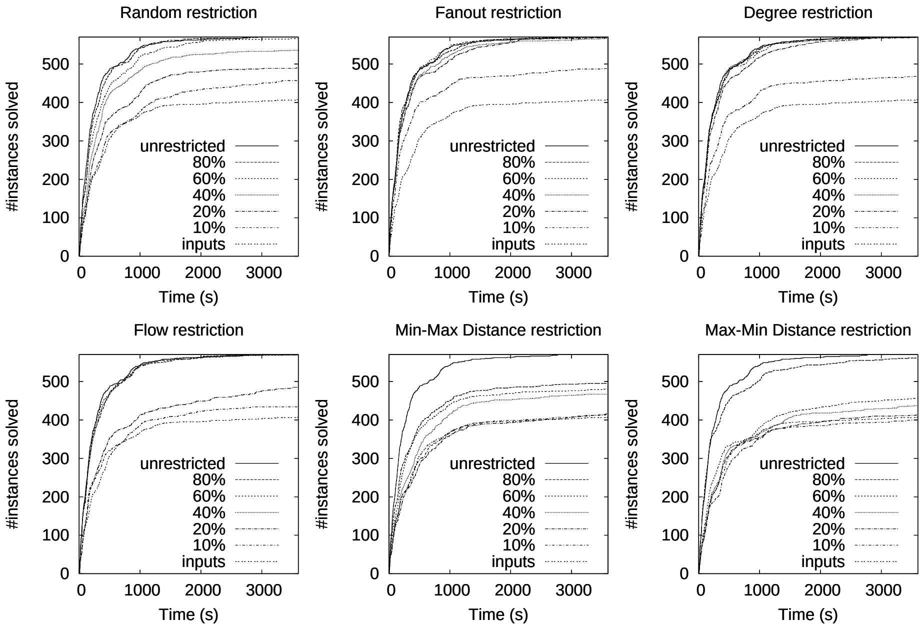

Fig. 6. Cumulative number of solved instances for the structural branching restrictions

5 Conclusions

We present an extensive experimental evaluation on 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. We show that by restricting the set of branchable variables to input variables, the effectiveness of the clause learning bound VSIDS heuristic and conflict clauses weakens: the best variables by the VSIDS heuristic are typically not branchable, and the average length of conflict clauses grows, resulting in the fact that the solver makes less decision per time unit. Applying schemes for allowing branching additionally on CNF variables other than inputs based on structural properties, we show that branching can by restricted rather heavily without losing the efficiency of the original unrestricted branching solves. The structural property based on which branching is restricted has an effect on efficiency. However, it seems unlikely that restricted branching could on its own make modern clause learning solvers more efficient in general. One interesting direction of further study 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 [39].

Acknowledgements. Financial support of HeCSE Graduate School, Academy of Finland (grant #211025), Emil Aaltonen Foundation, and the Technological Foundation TES is gratefully acknowledged. The author thanks T. Junttila and I. Niemelä for many insightful discussions.

References

1. Davis, M., Putnam, H.: A computing procedure for quantification theory. JACM **7**(3) (1960) 201–215
2. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. CACM **5**(7) (1962) 394–397
3. Hooker, J.N., Vinay, V.: Branching rules for satisfiability. JAR **15**(3) (1995) 359–383
4. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: IJCAI, Morgan Kaufmann (1997) 366–371
5. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC, ACM (2001) 530–535
6. Bacchus, F.: Enhancing Davis Putnam with extended binary clause reasoning. In: AAAI, AAAI Press (2002) 613–619
7. Li, C.M.: Equivalent literal propagation in Davis-Putnam procedure. DAM **130**(2) (2003) 251–276
8. Heule, M., van Maaren, H.: Aligning CNF- and equivalence-reasoning. In: SAT 2004 Selected Papers. Volume 3542 of LNCS., Springer (2005) 145–156
9. Bayardo, R.J., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: AAAI, AAAI Press (1997) 203–208
10. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: AAAI, AAAI Press (1998) 431–437
11. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5) (1999) 506–521
12. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust SAT-solver. In: DATE, IEEE (2002) 142–149
13. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT 2003. Volume 2919 of LNCS., Springer (2004) 502–518
14. Beame, P., Kautz, H.A., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. JAIR **22** (2004) 319–351
15. Heule, M., Dufour, M., van Zwieten, J., van Maaren, H.: March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In: SAT 2004 Selected Papers. Volume 3542 of LNCS., Springer (2005) 345–359
16. Anbulagan, Slaney, J.: Lookahead saturation with restriction for SAT. In: CP. Volume 3709 of LNCS., Springer (2005) 727–731
17. Huang, J.: The effect of restarts on the efficiency of clause learning. In: IJCAI, AAAI Press (2007) 2318–2323
18. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: DAC, ACM (1999) 317–320
19. Kautz, H.A., Selman, B.: Planning as satisfiability. In: ECAI, Wiley (1992) 359–363

20. Giunchiglia, E., Massarotto, A., Sebastiani, R.: Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In: AAAI, AAAI Press (1998) 948–953
21. Strichman, O.: Tuning SAT checkers for bounded model checking. In: CAV. Volume 1855 of LNCS., Springer (2000) 480–494
22. Giunchiglia, E., Maratea, M., Tacchella, A.: Dependent and independent variables in propositional satisfiability. In: JELIA. Volume 2424 of LNAI., Springer (2002) 296–307
23. Shi, J., Fey, G., Drechsler, R., Glowatz, A., Schlöffel, J., Hapke, F.: Experimental studies on SAT-based test pattern generation for industrial circuits. In: ASICON, IEEE (2005) 967–970
24. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in Boolean satisfiability solver. In: ICCAD, ACM (2001) 279–285
25. Marques-Silva, J.P.: The impact of branching heuristics in propositional satisfiability algorithms. In: EPIA. Volume 1695 of LNCS., Springer (1999) 62–74
26. Crawford, J.M., Baker, A.B.: Experimental results on the application of satisfiability algorithms to scheduling problems. In: AAAI, AAAI Press (1994) 1092–1097
27. Järvisalo, M., Junttila, T., Niemelä, I.: Unrestricted vs restricted cut in a tableau method for Boolean circuits. AMAI **44**(4) (2005) 373–399
28. Marques-Silva, J., Guerra e Silva, L.: Solving satisfiability in combinational circuits. IEEE Design & Test of Computers **20**(4) (2003) 16–21
29. Järvisalo, M., Junttila, T.: Limitations of restricted branching in clause learning. In: CP. LNCS, Springer (2007) To appear.
30. Liberatore, P.: Complexity results on DPLL and resolution. ACM TOCL **7**(1) (2006) 84–107
31. Williams, R., Gomes, C.P., Selman, B.: Backdoors to typical case complexity. In: IJCAI, Morgan Kaufmann (2003) 1173–1178
32. Kilby, P., Slaney, J., Thiébaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: AAAI, AAAI Press (2005) 1368–1373
33. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1995)
34. Velev, M., Bryant, R.: Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In: CHARME. Volume 1703 of LNCS., Springer (1999) 37–53
35. Pyhälä, T.: Factoring benchmarks for SAT-solvers (2004) `http://www.tcs.hut.fi/Software/genfacbm/`.
36. Järvisalo, M.: Equivalence checking multiplier designs (2007) SAT Competition 2007 benchmark description, `http://www.tcs.hut.fi/~mjj/benchmarks/`.
37. Jussila, T., Heljanko, K., Niemelä, I.: BMC via on-the-fly determinization. International Journal on Software Tools for Technology Transfer **7**(2) (2005) 89–101
38. Latvala, T., Biere, A., Heljanko, K., Junttila, T.A.: Simple bounded LTL model checking. In: FMCAD. Volume 3312 of LNCS., Springer (2004) 186–200
39. Thiffault, C., Bacchus, F., Walsh, T.: Solving non-clausal formulas with DPLL search. In: CP. Volume 3258 of LNCS., Springer (2004) 663–678