Limitations of Restricted Branching in Clause Learning

Matti Järvisalo and Tommi Junttila

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

Abstract. The techniques for making decisions, i.e., branching, play a central role in complete methods for solving structured CSP instances. In practice, there are cases when SAT solvers benefit from limiting the set of variables the solver is allowed to branch on to so called input variables. Theoretically, however, restricting branching to input variables implies a super-polynomial increase in the length of the optimal proofs for DPLL (without clause learning), and thus input-restricted DPLL cannot polynomially simulate DPLL. In this paper we settle the case of DPLL with clause learning. Surprisingly, even with unlimited restarts, input-restricted clause learning DPLL cannot simulate DPLL (even without clause learning). The opposite also holds, and hence DPLL and input-restricted clause learning DPLL are polynomially incomparable. Additionally, we analyse the effect of input-restricted branching on clause learning solvers in practice with various structural real-world benchmarks.

1 Introduction

Modern complete satisfiability (SAT) solvers provide an efficient way of solving various real-world problems as propositional satisfiability. Typical SAT solvers aimed at solving such structured problems are based on the conjunctive normal form (CNF) level *Davis-Putnam-Logemann-Loveland* procedure (DPLL) [1,2] and incorporate techniques such as *intelligent branching heuristics*, *randomisation* and *restarts* [3], and *clause learning* [4] for boosting search efficiency.

In SAT based approaches to structured problems such as bounded model checking [5] and automated planning [6], 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. Since irrelevant decisions may have an exponential effect on the running times of the solver, techniques for making decisions, i.e., branching, play a central role in complete SAT methods aimed at solving typically very large real-world problem instances. Empirical case studies [7,8,9,10] have shown that, in some cases, SAT solvers benefit from restricting the variables the solver is allowed to branch on to so called *input* (or *independent) variables*, corresponding to the input of the underlying system. Since the system behaviour is determined by its input, input-restricted branching DPLL remains complete. Intuitively, this drops the search space size from 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.

C. Bessiere (Ed.): CP 2007, LNCS 4741, pp. 348–363, 2007.
© Springer-Verlag Berlin Heidelberg 2007

From another point of view, one can investigate the *best-case* performance of SAT algorithms through *proof complexity* [11], by studying the relative power of their underlying inference systems (or *proof systems*) in terms of the shortest existing proofs in the systems. For two proof systems S,S', we say that S' *(polynomially) simulates* S if, for all infinite families $\{F_n\}$ of unsatisfiable CNF formulas, there is a polynomial that bounds for all F_n the length of the shortest proofs in S' w.r.t. the length of the shortest proofs in S. If S' simulates S and vice versa, then S and S' are *polynomially equivalent*. If S' cannot simulate S and vice versa, then S and S' are *incomparable*. From the practical point of view, if S' cannot simulate S, we know that any implementation of S' can suffer a notable decrease in efficiency compared to implementations of S. For example, through a formal characterisation CL of DPLL with clause learning, Beame et al. [12] show that CL can provide exponentially shorter proofs than DPLL, and thus DPLL cannot simulate CL.

Considering restricting branching in DPLL algorithms to input variables, a natural question to ask is *whether the power of the underlying inference systems of* DPLL *based solvers is affected by the input-restriction*. For DPLL without clause learning, this question is answered in [13]: input-restricted DPLL cannot simulate DPLL.

In this paper we settle the case of input-restricted CL: it turns out that input-restricted CL cannot simulate CL. This implies that all implementations of clause learning DPLL, even with optimal heuristics, have the potential of suffering a notable efficiency decrease if branching is restricted to input variables. In fact, we show that even with unlimited restarts and the ability to create conflicts at will, input-restricted CL cannot even simulate the basic DPLL *without clause learning*. This is surprising, since the unrestricted version of this variant of CL can efficiently simulate general resolution [12], being thus very powerful compared to DPLL. Additionally, we evaluate the effect of input-restricted branching on clause learning with various structural real-world benchmarks, and explain why branching restrictions are difficult to apply with typical clause learning search techniques.

As preliminaries, in Sect. 2 we define Boolean circuits, which we use for representing structural formulas, and discuss the close relation of circuits and CNF formulas. We then review the Resolution proof system and characterisations of DPLL and CL, and discuss known results concerning their relative efficiency (Sect. 3). The main theoretical and experimental contributions of this paper are presented in Sect. 4–5.

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 CNF formula by introducing additional variables for the sub-formulas of ϕ. *Boolean circuits* (see e.g. [14]) offer a natural way of presenting propositional formulas in a compact DAG-like structure with *sub-formula sharing*, which helps in lowering the number of additional variables needed. 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 the form $g := f(g_1, \ldots, g_n)$, where $g, g_1, \ldots, g_n \in G$ and $f : \{\mathbf{f}, \mathbf{t}\}^n \to \{\mathbf{f}, \mathbf{t}\}$ is a Boolean function, such 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 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. When convenient, we identify $\mathcal{C}$ with its underlying DAG. If $\langle g', g\rangle \in E(\mathcal{C})$, then g' is a *child* of g and g is a *parent* of g'. For any $g \in G$, 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) truth assignment for $\mathcal{C}$ is a (partial) function $\tau : G \to \{\mathbf{f}, \mathbf{t}\}$. A truth 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 truth 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 $\langle g, v\rangle \in \tau$. A truth assignment τ' *satisfies* $\mathcal{C}^\tau$ if (i) τ' is consistent with $\mathcal{C}$, and (ii) $\tau' \supseteq \tau$. If some truth assignment satisfies $\mathcal{C}^\tau$ then $\mathcal{C}^\tau$ is *satisfiable* and otherwise *unsatisfiable*.

For notational convenience, when well-defined, the *join* of constrained circuits $\mathcal{A}^\tau = \langle \mathcal{A}, \tau\rangle$ and $\mathcal{B}^\theta = \langle \mathcal{B}, \theta\rangle$ is $\mathcal{A}^\tau \cup \mathcal{B}^\theta := \langle \mathcal{A} \cup \mathcal{B}, \tau \cup \theta\rangle$. Without loss of generality, we restrict the set of Boolean functions available as gate types to

(i) $\text{NOT}(v)$ is $\mathbf{t}$ iff v is $\mathbf{f}$,

(ii) $\text{OR}(v_1, \ldots, v_n)$ is $\mathbf{t}$ iff at least one of $v_1, \ldots, v_n$ is $\mathbf{t}$,

(iii) $\text{AND}(v_1, \ldots, v_n)$ is $\mathbf{t}$ iff all $v_1, \ldots, v_n$ are $\mathbf{t}$, and

(iv) $\text{XOR}(v_1, v_2)$ is $\mathbf{t}$ iff exactly one of v_1, v_2 is $\mathbf{t}$.

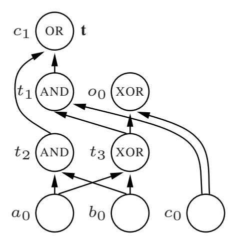

Fig. 1. A constrained circuit

As an example, Fig. 1 shows a Boolean circuit for a full-adder with the carry-out bit c_1 constrained to $\mathbf{t}$. Formally, this constrained circuit is $\langle \mathcal{C}, \tau\rangle$, where $\mathcal{C} = \{c_1 := \text{OR}(t_1, t_2),\ t_1 := \text{AND}(t_3, c_0),\ o_0 := \text{XOR}(t_3, c_0),\ t_2 := \text{AND}(a_0, b_0),\ t_3 := \text{XOR}(a_0, b_0)\}$ and $\tau = \{\langle c_1, \mathbf{t}\rangle\}$.

2.1 From Circuits to CNF, and CNF Formulas as Circuits

Given a Boolean variable x, there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\overline{x}$. As usual, we identify $\overline{\overline{x}}$ with x. A *clause* is a disjunction of distinct literals and a CNF formula is a conjunction of clauses. When convenient, we view a clause as a finite set of literals and a CNF formula as a finite set of clauses. The sets of variables appearing as positive and negative literals in a CNF F are denoted by $\mathsf{vars}^+(F)$ and $\mathsf{vars}^-(F)$, respectively, and the set of variables by $\mathsf{vars}(F)$; for a clause C, $\mathsf{vars}^+(C)$, $\mathsf{vars}^-(C)$, and $\mathsf{vars}(C)$ are defined similarly.

Given a CNF formula F, a (partial) *assignment* for F is a (partial) function $\tau : \mathsf{vars}(F) \to \{\mathbf{t}, \mathbf{f}\}$. With slight abuse of notation, if $\tau(x) = v$, then $\tau(\overline{x}) = \neg v$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$. A clause is satisfied by τ if it contains at least one literal l such that $\tau(l) = \mathbf{t}$. An assignment τ *satisfies* F if it satisfies every clause in F. A formula is *satisfiable* if there is an assignment that satisfies it, and *unsatisfiable* otherwise. We apply the stardard "Tseitin translation" to map each constrained Boolean circuit $\langle \mathcal{C}, \tau\rangle$ into an equi-satisfiable CNF formula $\mathsf{cnf}(\langle \mathcal{C}, \tau\rangle)$. To obtain a small CNF formula, the

Table 1. Translating a constrained Boolean circuit $\langle \mathcal{C}, \tau \rangle$ to the CNF formula $\mathsf{cnf}(\langle \mathcal{C}, \tau \rangle)$

circuit $\langle \mathcal{C}, \tau \rangle$	clauses in $\mathsf{cnf}(\langle \mathcal{C}, \tau \rangle)$
$g := \text{NOT}(g_1) \in \mathcal{C}$	$\{\bar{x}_g, \bar{x}_{g_1}\}$, $\{x_g, x_{g_1}\}$
$g := \text{OR}(g_1, \ldots, g_n) \in \mathcal{C}$	$\{\bar{x}_g, x_{g_1}, \ldots, x_{g_n}\}$, $\{x_g, \bar{x}_{g_1}\}, \ldots, \{x_g, \bar{x}_{g_n}\}$
$g := \text{AND}(g_1, \ldots, g_n) \in \mathcal{C}$	$\{\bar{x}_g, x_{g_1}\}, \ldots, \{\bar{x}_g, x_{g_n}\}$, $\{x_g, \bar{x}_{g_1}, \ldots, \bar{x}_{g_n}\}$
$g := \text{XOR}(g_1, g_2) \in \mathcal{C}$	$\{\bar{x}_g, \bar{x}_{g_1}, \bar{x}_{g_2}\}$, $\{\bar{x}_g, x_{g_1}, x_{g_2}\}$, $\{x_g, \bar{x}_{g_1}, x_{g_2}\}$, $\{x_g, x_{g_1}, \bar{x}_{g_2}\}$
$\langle g, \mathbf{t} \rangle \in \tau$	$\{x_g\}$
$\langle g, \mathbf{f} \rangle \in \tau$	$\{\bar{x}_g\}$

idea is to introduce a variable x_g for each gate g in the circuit, and then to describe the functionality of each gate with a linear number of clauses (Table 1).

Any CNF formula $F = \{C_1, \ldots, C_k\}$ can naturally be seen as a Boolean circuit. Basically, F is a Boolean circuit with an AND of ORs which represent the clauses. Formally, $\mathsf{circuit}(F) := \langle \mathcal{C}, \tau \rangle$ is defined by associating an input gate g_x with each $x \in \mathsf{vars}(F)$, a NOT-gate $g_{\bar{x}}$ with each $x \in \mathsf{vars}^-(F)$, an OR-gate g_{C_i} with each clause $C_i \in F$, an AND-gate g_F with F, and defining $\tau = \{\langle g_F, \mathbf{t} \rangle\}$ and

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

3 Resolution, DPLL, and CL with Variants

We now review proof systems for CNF formulas, namely, *Resolution*, and characterisations of DPLL and CL [12] (DPLL with clause learning). We will apply these in Sect. 4.

3.1 Resolution

The well-known Resolution proof system (RES) is based on the *resolution rule*. Let C, D be clauses, and x a Boolean variable. The resolution rule lets us derive the clause $C \cup D$ from the clauses $\{x\} \cup C$ and $\{\bar{x}\} \cup D$ by *resolving on* x. A RES *proof (for the unsatisfiability) of a CNF formula* F is a sequence of clauses $\pi = (C_1, C_2, \ldots, C_m = \emptyset)$, where each C_i, $1 \le i \le m$, is either (i) a clause in F (an *initial clause*), or (ii) derived with the resolution rule from two clauses C_j, C_k where $1 \le j, k < i$ (a *derived clause*). The *length* of π is m, the number of clauses occurring in it.

Many *refinements of* Resolution, in which the structure of RES proofs is restricted, have been proposed and studied. Here of particular interest is *Tree-like Resolution* (T-RES) that requires the refutations to be representable as trees. This implies that a derived clause, if subsequently used multiple times in the refutation, must be derived anew each time starting from initial clauses.

Superpolynomial lower bounds on proof length in RES have been shown for various families of CNF formulas. Among the most studied such families is the *pigeon-hole principle*, which states that there is no injective mapping from an m-element set into an n-element set if $m > n$ (i.e., m pigeons cannot sit in less than m holes so that every pigeon has its own hole). We consider the case $m = n + 1$ encoded as the CNF formula

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

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

Theorem 1 (Haken [15]). *There is no polynomial length* RES *proof of* PHP_n^{n+1}.

It is also known that T-RES is a *proper* refinement of RES. This originates from the facts that *regular resolution* cannot simulate RES [16], and T-RES in turn cannot simulate regular resolution [17].

Corollary 1 (of [16,17]). T-RES *cannot polynomially simulate* RES.

3.2 DPLL

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 on $\mathsf{vars}(F)$ through *branching* and *unit propagation* (UP). By branching the current partial assignment τ is extended with $\tau(x) = v$, $v \in \{\mathbf{f}, \mathbf{t}\}$, for some unassigned variable x. Unit propagation refers to the process of immediately applying the *unit clause rule* with which the current partial assignment τ is extended with $\tau(l) = \mathbf{t}$ if there is a clause $\{l_1, \ldots, l_k, l\} \in F$ such that $\tau(l_i) = \mathbf{f}$ for each $1 \leq i \leq k$. A branch is extended until (i) there is a clause $C \in F$ for which $\tau(l) = \mathbf{f}$ for each literal $l \in C$, or (ii) τ satisfies F. In case (i), τ is *conflicting* with the particular clause, and DPLL *backtracks* to the last branching decision whose other branch has not been tried yet, and flips the particular decision in τ. A DPLL search terminates when either a satisfying assignment is found, or when all possible branches have been covered, in which case F is determined as unsatisfiable.

As a proof system, the strength of DPLL does not depend on whether UP is applied. Any application of the unit clause rule on a clause C can be simulated by branching on the remaining unassigned literal $l \in C$; assigning l a conflicting value by branching causes immediately backtracking. It is well-known that DPLL and T-RES can polynomially simulate each other; one can show that for any unsatisfiable CNF formula, with UP seen as branching, the branches tried by DPLL correspond one-to-one with the paths of a T-RES proof with the conflicting clauses as leafs.

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

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

Implication graphs capture naturally the ways of deriving all implied literals from decision literals by UP.

Definition 1. *The* implication graph G *at a given stage of* DPLL *is a directed graph with edges labeled with sets of clauses. An implication graph is constructed as follows.*

1. *Create a node for each decision literal, labeled with that literal.*
2. *While there is a clause* $C = \{l_1, \ldots, l_k, l\}$ *such that* $\bar{l}_1, \ldots, \bar{l}_k$ *label nodes in* G*,*
   (a) Add a node labeled l *if not already in* G*.*
   (b) Add edges $\langle l_i, l\rangle$ *for* $1 \le i \le k$*, if not already present.*
3. *Add a special node* Λ *to* G*. For any variable* x *with both labels* x *and* $\bar{x}$ *in* G*, add edges* $\langle x, \Lambda\rangle$ *and* $\langle \bar{x}, \Lambda\rangle$*. Any such* $x, \bar{x}$ *are* conflict literals*, and the variable* x *is a* conflict variable.

An implication graph contains a conflict if it contains a conflict variable; DPLL has a conflict at a given stage if the implication graph at the stage contains a conflict.

3.3 Clause Learning

Most state-of-the-art complete SAT solvers today apply DPLL enhanced with conflict analysis [4], resulting in Clause Learning (CL). Like the basic DPLL, CL performs branching and UP until a conflict is reached. If this happens without any branching, CL determines the formula F 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 CL continues by backtracking as DPLL does, or can backjump to an earlier decision level that "caused" the conflict (as discussed in more detail below).

At a given stage of a CL search procedure, 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 one way of reaching the conflict at hand form the decision variables by using UP on known clauses.

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

1. H *contains* Λ *and exactly one conflict literal pair* $x, \bar{x}$*.*
2. *All nodes in* H *have a path to* Λ*.*
3. *Every node* $l \in V \setminus \{\Lambda\}$ *either corresponds to a decision literal or has precisely the nodes* $\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_k$ *as predecessors where* $\{l_1, l_2, \ldots, l_k, l\}$ *is a known clause.*

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

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

A clause learning proof (or CL proof) under a learning scheme is a CL search tree using that learning scheme. The length of the proof is the number of branching decisions. The proof system CL consists of CL proofs under any learning scheme.

While the practical efficiency gains of implementing clause learning into DPLL based algorithms are well-established, the first formal study on the power of clause learning is [12]: CL can provide exponentially shorter proofs than T-RES, and thus

Corollary 2 (of Fact 1 and [12]). DPLL *cannot polynomially simulate* CL.

Typically implemented clause learning schemes are based on *unique implication points* (UIPs) [4]. A UIP of a conflict graph is a node u on the maximal decision level d such that all paths from the decision variable x at level d to Λ go through u. Such a u always exists, since x satisfies this condition; intuitively u is a *single* reason for the conflict at level d. Thus one can always choose a conflict cut that results in a conflict clause with a UIP as the only variable from the maximal decision level. Such a conflict clause causes the value of the UIP to be immediate flipped when backtracking. Furthermore, UIP learning enables *conflict-driven backtracking* (or *backjumping*), in which DPLL 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 the UIP closest to Λ is chosen. Different learning schemes are evaluated in [18].

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 0. The clauses learned so far remain known after the restart. Intuitively, restarts help in escaping from getting stuck in hard-to-prove subformulas. In practice, the choice of when and how often to restart is again part of the strategy of a solver. When any number of restarts are allowed during search, CL has *unlimited restarts*.

Beame et al. [12] define CL-- as CL with branching allowed also on literals already set at the current stage of DPLL. Although being non-typical in practice, this enables creating immediate conflicts at will. Although it is not known whether CL can simulate RES, it has been shown that this is true for CL-- using restarts.

Theorem 2 (Beame et al. [12]). RES *and* CL-- *with unlimited restarts and any non-redundant learning scheme are polynomially equivalent.*

In the following, we will explicitly mention when restarts are allowed.

3.4 Input-Restricted Branching DPLL and CL

In structural application domains of SAT solvers, such as planning and bounded model checking of hardware and software, Boolean circuits offer a natural presentation form for the problem descriptions. Typically, such problems are based on a transition relation, where the behaviour of the underlying system is dependent solely on the *input* of the system. In the Boolean circuit encoding $\langle \mathcal{C}, \tau \rangle$, the input is represented by the set of input gates (sometimes called *independent variables*) of the circuit, $\mathsf{inputs}(\mathcal{C})$. Since the circuit can be evaluated when all gates in $\mathsf{inputs}(\mathcal{C})$ have values, branching in DPLL *with unit propagation* can be restricted to the variables associated with $\mathsf{inputs}(\mathcal{C})$—denoted by $\mathsf{DPLL}_{\mathsf{inputs}}$ and $\mathsf{CL}_{\mathsf{inputs}}$ for clause learning—without losing completeness. Intuitively, the idea is that since $|\mathsf{inputs}(\mathcal{C})|$ is often much less than the total amount $|G|$

of gates in $\mathcal{C}$, search space size is reduced from $2^{|G|}$ to $2^{|\mathsf{inputs}(\mathcal{C})|}$, where $|\mathsf{inputs}(\mathcal{C})| << |G|$. From the view of proof complexity, however, in [13] a formal study on the effect of restricting branching in DPLL (without clause learning) to inputs reveals that this weakens the proof system considerably.

Theorem 3 (Järvisalo et al. [13]). $\mathsf{DPLL}_{\mathsf{inputs}}$ *cannot polynomially simulate* DPLL.

The rest of the paper is dedicated to investigating the effect of restricting branching to inputs in the case of clause learning, which is posed as an open question in [13].

4 Separating Input-Restricted and Unrestricted CL

We will now consider the relative power of input-restricted and unrestricted CL and DPLL. This will result in a refined relative efficiency hierarchy of DPLL and CL (Fig. 3).

Since the cnf translation associates a variable for each gate in a circuit, when appropriate we will use the term "(e.g., branch on, set value to) gate g" when referring to the variable x_g associated with g in the CNF translation of the circuit. Correspondingly, a DPLL or CL proof of a constrained circuit $\mathcal{C}^\tau$ means a proof of the translation $\mathsf{cnf}(\mathcal{C}^\tau)$.

Lemma 1. *There is an infinite family* $\{\mathcal{C}_n^\tau\}$ *of constrained Boolean circuits for which* DPLL *has exponentially longer minimal proofs than* $\mathsf{CL}_{\mathsf{inputs}}$.

Proof. Take any infinite family $\{F_n\}$ of CNF formulas that is a witness of Corollary 2. Define the family of Boolean circuits $\{\mathsf{circuit}(F) \mid F \in \{F_n\}\}$. The formula resulting from UP on $\mathsf{cnf}(\mathsf{circuit}(F))$ without branching corresponds to the result of unit propagation on F without branching. Thus DPLL will only branch on the variables in $\mathsf{cnf}(\mathsf{circuit}(F))$ that are associated with the input gates of $\mathsf{circuit}(F)$. Thus $\mathsf{CL}_{\mathsf{inputs}}$ can simulate CL on $\mathsf{cnf}(\mathsf{circuit}(F))$, and the claim follows by Corollary 2. □

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

To highlight the strength of clause learning even when branching is restricted to input gates, we now give an example of a family $\{\text{XOR-UNSAT}_n\}$ of Boolean circuits on which $\mathsf{CL}_{\mathsf{inputs}}$ can simulate CL, although $\mathsf{DPLL}_{\mathsf{inputs}}$ cannot simulate DPLL on the family. The circuit $\text{XOR-UNSAT}_n := \text{UNSAT} \cup \langle \text{XOR}_n^a \cup \text{XOR}_n^b, \emptyset\rangle$ consists of two parts: (i) the constant size circuit $\text{UNSAT} := \mathsf{circuit}(\{\{a,b\},\{a,\bar{b}\},\{\bar{a},b\},\{\bar{a},\bar{b}\}\})$ and (ii) two copies (for a and b, $\rho \in \{a,b\}$) of the circuit structure

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

XOR-UNSAT_2 is shown in Fig. 2. Now, since UP will result in a conflict in the UNSAT subcircuit for any value of the gate a, XOR-UNSAT_n yields a trivial (constant length) proof in DPLL. It is also easy to see that minimal length proofs of XOR-UNSAT_n are exponential w.r.t. n in $\mathsf{DPLL}_{\mathsf{inputs}}$. Due to the structure of XOR_n, in order to propagate a value for the gate a or b, $\mathsf{DPLL}_{\mathsf{inputs}}$ has to branch on all of the inputs in the corresponding XOR_n^ρ subcircuit. With the backtracking process of DPLL this implies that minimal length $\mathsf{DPLL}_{\mathsf{inputs}}$ proofs of XOR-UNSAT_n are exponential w.r.t. n.

However, $\mathsf{CL}_{\mathsf{inputs}}$ can produce linear length proofs on the family. Let $\mathsf{CL}_{\mathsf{inputs}}$ branch according to the sequence $(x^a_{n,1} = \mathbf{f}, \ldots, x^a_{n,n} = \mathbf{f})$. After this, UP cannot still propagate any values. Then branch with $x^a_{n,n+1} = \mathbf{f}$. Now UP sets values for all $x^a_{i,j}$, without a conflict. The values for $x^a_{1,1}$ and $x^a_{1,2}$ propagate a value for a, which then propagates a conflict at a gate in UNSAT. Notice that $x^a_{1,1}$ and $x^a_{1,2}$ are the *only* reasons for the value of a. In *any* conflict graph associated with the branching sequence $(x^a_{n,1} = \mathbf{f}, \ldots, x^a_{n,n+1} = \mathbf{f})$, a is an UIP, and, furthermore, constitutes a reason for the conflict on its own. Hence $\mathsf{CL}_{\mathsf{inputs}}$ can learn as a unit clause the opposite value of a, and backjump to the decision level zero. This opposite value will then propagate a contradiction without branching, and $\mathsf{CL}_{\mathsf{inputs}}$ terminates. It is interesting to notice how $\mathsf{CL}_{\mathsf{inputs}}$ can branch on $(x^a_{n,1} = \mathbf{f}, \ldots, x^a_{n,n+1} = \mathbf{f})$ and still avoid backtracking on these decisions since there is the *bottleneck* at gate a due to the construction of XOR-UNSAT$_n$. This shows the power of clause learning with conflict-driven backtracking due to its ability to backjump over an exponential size search space by detecting small locally inconsistent subformulas. With this intuition, it is evident that the results in [13] on the power $\mathsf{DPLL}_{\mathsf{inputs}}$ w.r.t. DPLL cannot be directly adopted for proving the analogous result for $\mathsf{CL}_{\mathsf{inputs}}$.

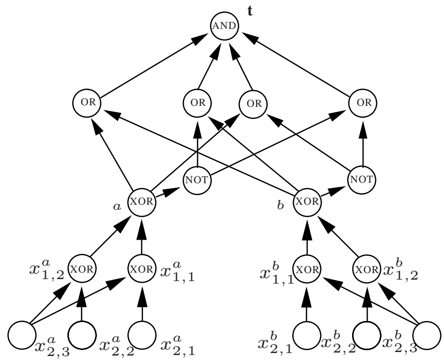

Fig. 2. XOR-UNSAT$_n$ for $n = 2$

Although $\mathsf{CL}_{\mathsf{inputs}}$ can simulate CL on this specific family, this is generally not the case. In fact, it turns out that $\mathsf{CL}_{\mathsf{inputs}}$ *cannot even simulate* DPLL, as detailed next. We will apply the concept of *redundant gates in constraint Boolean circuits*.

Definition 3. *A gate g in a constrained Boolean circuit $\langle\langle G, E\rangle, \tau\rangle$ is* redundant *if (i) g is unconstrained, and (ii) g is not a descendant of any constrained gate g' in $\langle G, E\rangle$.*

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

Lemma 2. *Redundant gates do not occur in any conflict graph at any stage of* $\mathsf{CL\text{--}}_{\mathsf{inputs}}$ *whether or not restarts are allowed.*

Proof. For a constrained circuit $\langle\langle G, E\rangle, \tau\rangle$, a *subcircuit* $\langle\langle G', E'\rangle, \tau'\rangle$ induced by $G' \subseteq G$ is $E' = \{\langle g, g'\rangle \in E \cap (G' \times G')\}$ and $\tau' = \{\langle g, \epsilon\rangle \in \tau \mid g \in G'\}$.

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

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

Redundant gates can be removed from any constrained Boolean circuit without affecting its satisfiability. However, they may have an effect on the length of minimal proofs. Cook [19] gives a way of introducing a polynomial number of clauses which can be interpreted as redundant gates to $\mathsf{circuit}(\mathrm{PHP}^{n+1}_n)$ so that, contrarily to $\mathsf{circuit}$ (PHP^{n+1}_n), the extended circuit yields polynomial length proofs in RES. As a circuit structure, this *extension* is defined as $\mathrm{EXT}_n := \bigcup_{l=1}^{n} \mathrm{EXT}^l$, where

$$\mathrm{EXT}^l := \bigcup_{i=1}^{l} \bigcup_{j=1}^{l-1} \{e^l_{i,j} := \text{OR}(e^{l+1}_{i,j}, o^l_{i,j}),\ o^l_{i,j} := \text{AND}(e^{l+1}_{i,l}, e^{l+1}_{l+1,j})\},$$

and each $e^n_{i,j}$ is the input gate $g_{p_{i,j}}$ associated with the variable $p_{i,j}$ in PHP^{n+1}_n. By [19] we immediately have a polynomial length RES proof [1] $\pi = (C_1, \ldots, C_m = \emptyset)$ of $\mathsf{cnf}(\mathsf{circuit}(\mathrm{PHP}^{n+1}_n) \cup \langle \mathrm{EXT}_n, \emptyset \rangle)$. Using π, we define the construct

$$\mathrm{E}(\pi) := \bigcup_{i=2}^{m-1} \{h_i := \text{AND}(g_{C_i}, h_{i-1})\} \cup \bigcup_{i=1}^{m-1} \{\hat{g} := \text{NOT}(g) \mid x_g \in \mathsf{vars}^-(C_i)\}$$
$$\bigcup_{i=1}^{m-1} \{g_{C_i} := \text{OR}(\alpha(l_{i,1}), \ldots, \alpha(l_{i,k_i})) \mid C_i = \{l_{i,1}, \ldots, l_{i,k_i}\}\}$$

where h_1 is the gate g_{C_1}, $\alpha(x_g) = g$, and $\alpha(\bar{x}_g) = \hat{g}$. The construct encodes π in a way that allows polynomial length DPLL proofs of $\mathrm{EPHP}_n = \mathsf{circuit}(\mathrm{PHP}^{n+1}_n) \cup \langle \mathrm{EXT}_n, \emptyset \rangle \cup \langle \mathrm{E}(\pi), \emptyset \rangle$, while there is no polynomial length $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ proof of EPHP_n. Intuitively this is because $\mathrm{E}(\pi)$ allows DPLL to "verify" the resolution proof of PHP^{n+1}_n extended with EXT_n step-by-step, while $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ cannot make use of the redundant gates of EXT_n and $\mathrm{E}(\pi)$.

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

Proof. A polynomial length DPLL proof of EPHP_n is witnessed by the branching sequence $(h_1 = \mathbf{f}, h_2 = \mathbf{f}, \ldots, h_{m-1} = \mathbf{f})$, as detailed next. By induction on i, we will

[1] Due to space constraint, we do not give π explicitly. Intuitively, EXT^l allows reducing PHP^{l+1}_l to PHP^l_{l-1} with a polynomial number of resolution steps. For more details, see the report version [20].

show that, if $h_1 = \mathbf{t}, \ldots, h_{i-1} = \mathbf{t}$, then branching with $h_i = \mathbf{f}$ results in a conflict by UP, and hence immediately setting $h_i = \mathbf{t}$.

The base case. The gate $h_1 = g_{C_1}$ represents the first clause C_1 in π, and C_1 must belong to $\mathsf{cnf}(\mathsf{circuit}(\mathrm{PHP}_n^{n+1}) \cup \langle \mathrm{EXT}_n, \emptyset \rangle)$. As C_1 is a result of applying the cnf translation to a gate g in $\mathsf{circuit}(\mathrm{PHP}_n^{n+1}) \cup \langle \mathrm{EXT}_n, \emptyset \rangle$ (which is part of EPHP_n), setting $h_1 = \mathbf{f}$ will result in a conflict after UP because the functional definition or the constraint of the gate g is violated. For example, if $g := \textsc{or}(g_1, g_2)$ and $C_1 = \{x_g, \bar{x}_{g_1}\}$, then $h_1 = g_{C_1} := \textsc{or}(g, \hat{g_1})$, $\hat{g_1} := \textsc{not}(g_1)$, and the assignment $h_1 = \mathbf{f}$ will propagate $g = \mathbf{f}$ and $g_1 = \mathbf{t}$, violating the definition of g and thus resulting in a conflict.

Now assume as the induction hypothesis that we have $h_{i'} = \mathbf{t}$ for all $1 \leq i' < i$. Recall that $h_i := \textsc{and}(g_{C_i}, h_{i-1})$. By branching with $h_i = \mathbf{f}$, UP sets $g_{C_i} = \mathbf{f}$ by the induction hypothesis. If the ith clause C_i in π belongs to $\mathsf{cnf}(\mathsf{circuit}(\mathrm{PHP}_n^{n+1}) \cup \langle \mathrm{EXT}_n, \emptyset \rangle)$, branching on $g_{C_i} = \mathbf{f}$ will result in a conflict after UP as in the base case. Otherwise C_i has been derived from two clauses, $C_j = C_j' \cup \{x_g\}$ and $C_k = C_k' \cup \{\bar{x}_g\}$, in π for $1 \leq j, k < i$, by resolving on a variable x_g. By the induction hypothesis we have $h_j = \mathbf{t}$ and $h_k = \mathbf{t}$, and thus $g_{C_j} = \mathbf{t}$ and $g_{C_k} = \mathbf{t}$ by UP. On the other hand, as $g_{C_i} = \mathbf{f}$, all the gates corresponding to the literals in $C_j' \cup C_k'$ are assigned to $\mathbf{f}$ by UP, implying that UP will assign both $g = \mathbf{t}$ and $g = \mathbf{f}$ as $g_{C_j} = g_{C_k} = \mathbf{t}$. Thus a conflict is reached, closing the branch $h_i = \mathbf{f}$, and $h_i = \mathbf{t}$ is set by backtracking.

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

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

Directly by Lemmas 1 and 3 we have

Theorem 4. $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ *(with or without restarts) and* DPLL *are incomparable.*

Corollary 4. $\mathsf{CL}\text{--}_{\mathsf{inputs}}$ *with unlimited restarts cannot polynomially simulate* CL.

Remark 1. We use redundant gates in the EPHP_n construction for simplicity of the proof of Lemma 3; by a simple modification of EPHP_n one can construct as a witness for Lemma 3 a constrained circuit with no redundant gates and a single output as the only constrained gate.

Remark 2. Since redundant gates can be removed from constrained Boolean circuits without affecting the sets of satisfying assignments, such gates are typically removed in practice before the CNF translation by so called *cone-of-influence reduction*. However, as witnessed by EPHP_n in Lemma 3, applying cone-of-influence can have a drastic negative effect on the minimal length proofs. It is especially interesting to notice that DPLL solvers with full one-step lookahead can detect the small proofs of EPHP_n witnessed by the branching sequence $(h_1 = \mathbf{f}, h_2 = \mathbf{f}, \ldots, h_{n-1} = \mathbf{f})$.

Figure 3 gives a refined relative efficiency hierarchy for the proof systems considered in this paper. An arrow without a slash from system S to S' means that S can polynomially

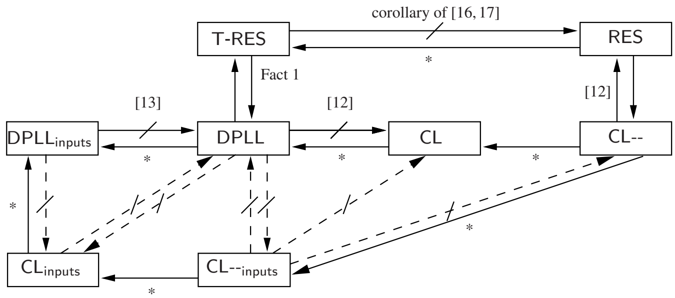

Fig. 3. A refined relative efficiency hierarchy for the proof systems considered in this paper

simulate S', and with a slash that S cannot simulate S'. Arrows labeled with $*$ are due to trivial subsumption. The new results, detailed in the following, are represented by dashed arrows. Disregarding transitivity of the results, missing arrows represent questions which are open to the best of our knowledge.

5 Experiments

We evaluate the effect of input-restriction on the functionality of modern clause learning solver techniques. The benchmark set used in the experiments consists of instances from various application domains, for which Boolean circuits offer a natural representation form: super-scalar processor verification [21], integer factorisation based on hardware multipliers [22], equivalence checking of hardware multipliers [23], bounded model checking (BMC) for deadlocks in asynchronous parallel systems as labelled transition systems (LTS) [24], and linear temporal logic (LTL) BMC of finite state systems with a linear encoding [25]. We use standard PCs with 2-GHz AMD 3200+ processors and 2 GBs of memory running Linux, with a timeout of 1 hour and a memory limit of 1 GB.

For solving the Boolean circuit instances, we apply BCMinisat[2] (version 0.26), which we have modified in order to restrict branching to input variables. BCMinisat is a Boolean circuit front-end for the successful clause learning SAT solver Minisat [26] (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, normalising the circuit into a form which can be translated into CNF applying a standard translation in the style of cnf defined in Table 1. BCMinisat feeds the resulting CNF translations and the input-restriction to Minisat, which then solves the CNF. For each circuit, we obtain 15 CNF instances by permuting the CNF variable numbering.

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

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

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 factorisation										
`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)	-	-

5.1 Results

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 multiplication 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, we notice that 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.

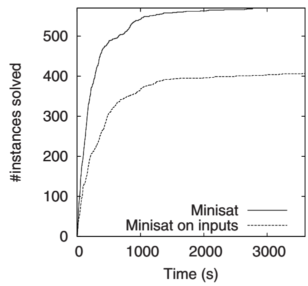

Fig. 4. Solved instances

Figure 4 gives a cumulative plot of the number of solved instances, showing a drastic decrease in performance for the input-restriction. The effect of input-restriction varies depending on whether unsatisfiable or satisfiable instances are considered (leftmost and middle plots in Fig. 5). For the unsatisfiable instances the plot correlates well with Corollary 4, with timed out runs on the horizontal line. For satisfiable instances, there seems to be no clear winner, although when selecting from the relative small set of input variables, the probability of choosing a satisfying assignment is intuitively greater. A noticeable point is that, while BCMinisat$_{\mathsf{inputs}}$ makes less decisions, e.g, on the equivalence checking instances, unrestricted BCMinisat is at least as efficient as BCMinisat$_{\mathsf{inputs}}$ w.r.t. running times. Interestingly, this is due to the fact that unrestricted BCMinisat often *manages more decisions per second* (on the right in Fig. 5).

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

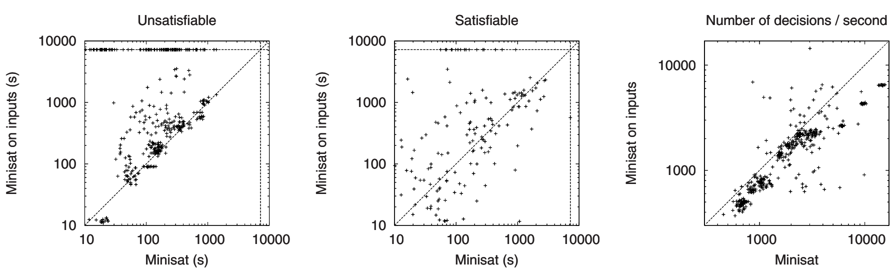

Fig. 5. Scatter plots: running times on unsatisfiable (left) and satisfiable (middle) instances; number of decisions / second (right)

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.

6 Conclusions

We investigate the effect of restricting branching in clause learning SAT solving on the efficiency of the underlying inference system from the view of proof complexity. Although the unrestricted version of the considered variant of clause learning can efficiently simulate general resolution, being thus very powerful compared to DPLL, we show the surprising result that input-restricted clause learning cannot even simulate the basic DPLL *without clause learning*. This implies that all implementations of clause learning DPLL, even with optimal heuristics, have the potential of suffering a notable efficiency decrease if branching is restricted to input variables. Notably, the results directly apply to SAT based approaches to solving Boolean combinations of more general constraints, for example, *Satisfiability Modulo Theories*, where the propagation mechanisms for the Boolean combinations can be seen as a form of unit propagation. The experimental evidence shows that by restricting branching the robustness of SAT solvers can decrease, and that input-branching does not go well with clause learning based heuristics of modern solvers.

Acknowledgements. The authors thank Ilkka Niemelä and Emilia Oikarinen for fruitful discussions. Järvisalo gratefully acknowledges the financial support of Helsinki Graduate School in Computer Science and Engineering, Academy of Finland (grant #211025), the Emil Aaltonen Foundation, and the Technological Foundation TES.

References

1. Davis, M., Putnam, H.: A computing procedure for quantification theory. JACM 7(3), 201–215 (1960)
2. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. CACM 5(7), 394–397 (1962)
3. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: AAAI, pp. 431–437. AAAI Press, Stanford, California, USA (1998)
4. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Comp. 48(5), 506–521 (1999)
5. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: DAC, pp. 317–320. ACM Press, New York (1999)
6. Kautz, H.A., Selman, B.: Planning as satisfiability. In: ECAI, pp. 359–363. Wiley, Chichester (1992)
7. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.Y.: Benefits of bounded model checking at an industrial setting. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 436–453. Springer, Heidelberg (2001)
8. Giunchiglia, E., Massarotto, A., Sebastiani, R.: Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In: AAAI, pp. 948–953. AAAI Press, Stanford, California, USA (1998)

9. Strichman, O.: Tuning SAT checkers for bounded model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
10. Giunchiglia, E., Maratea, M., Tacchella, A.: Dependent and independent variables in propositional satisfiability. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 296–307. Springer, Heidelberg (2002)
11. Cook, S.A., Reckhow, R.: On the relative efficiency of propositional proof systems. J. Symb. Logic 44, 36–50 (1977)
12. Beame, P., Kautz, H.A., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. JAIR 22, 319–351 (2004)
13. Järvisalo, M., Junttila, T., Niemelä, I.: Unrestricted vs restricted cut in a tableau method for Boolean circuits. AMAI 44(4), 373–399 (2005)
14. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1995)
15. Haken, A.: The intractability of resolution. TCS 39(2–3), 297–308 (1985)
16. Goerdt, A.: Regular resolution versus unrestricted resolution. SIAM J. Comp. 22(4), 661–683 (1993)
17. Urquhart, A.: The complexity of propositional proofs. B. Symb. Logic 1(4), 425–467 (1995)
18. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD, pp. 279–285 (2001)
19. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News 8(4), 28–32 (1976)
20. Järvisalo, M.: Impact of restricted branching on clause learning SAT solving. Research Report A107, Helsinki University of Technology, Laboratory for Theoretical Computer Science (2007), See
`http://www.tcs.hut.fi/Publications/`
21. Velev, M., Bryant, R.: Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 37–53. Springer, Heidelberg (1999)
22. Pyhälä, T.: Factoring benchmarks for SAT-solvers (2004),
`http://www.tcs.hut.fi/Software/genfacbm/`
23. Järvisalo, M.: Equivalence checking multiplier designs, SAT Competition 2007 benchmark description (2007),
`http://www.tcs.hut.fi/~mjj/benchmarks/`
24. Jussila, T., Heljanko, K., Niemelä, I.: BMC via on-the-fly determinization. International Journal on Software Tools for Technology Transfer 7(2), 89–101 (2005)
25. Latvala, T., Biere, A., Heljanko, K., Junttila, T.A.: Simple bounded LTL model checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 186–200. Springer, Heidelberg (2004)
26. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)