# Finding Small Backdoors in SAT Instances

Zijie Li and Peter van Beek

Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

**Abstract.** Although propositional satisfiability (SAT) is NP-complete, state-of-the-art SAT solvers are able to solve large, practical instances. The concept of backdoors has been introduced to capture structural properties of instances. A backdoor is a set of variables that, if assigned correctly, leads to a polynomial-time solvable sub-problem. In this paper, we address the problem of finding all small backdoors, which is essential for studying value and variable ordering mistakes. We discuss our definition of sub-solvers and propose algorithms for finding backdoors. We experimentally compare our proposed algorithms to previous algorithms on structured and real-world instances. Our proposed algorithms improve over previous algorithms for finding backdoors in two ways. First, our algorithms often find smaller backdoors. Second, our algorithms often find a much larger number of backdoors.

## 1 Introduction

Propositional satisfiability (SAT) is a core problem in AI. The applications are numerous and include software and hardware verification, planning, and scheduling. Even though SAT is NP-complete in general, state-of-the-art SAT solvers can solve large, practical problems with thousands of variables and clauses. To explain why current SAT solvers scale well in practice, Williams, Gomes, and Selman [13, 14] propose the concept of weak and strong backdoors to capture structural properties of instances. A weak backdoor is a set of variables for which there exists a value assignment that leads to a polynomial-time solvable sub-problem. For a strong backdoor, every value assignment should lead to a polynomial-time solvable sub-problem.

In this paper, we address the problem of finding all small backdoors in SAT instances. A small backdoor is a backdoor such that no proper subset is also a backdoor. This problem is important for studying problem hardness, which is generally represented as the time used or the number of nodes extended by a SAT solver. In addition, identifying all small backdoors is a first step to investigating how value and variable ordering mistakes affect the performance of backtracking algorithms—the ultimate goal of our research. A variable ordering heuristic can make a mistake by selecting a variable not in the appropriate backdoor. A value ordering heuristic can make a mistake by assigning the backdoor variable a value that does not lead to a polynomial sub-problem.

Backdoors are defined with respect to sub-solvers, which in turn can be defined algorithmically or syntactically. Algorithmically defined sub-solvers are polynomial-time techniques of current SAT solvers, such as unit propagation. Syntactically defined sub-solvers are polynomial-time tractable classes, such as 2SAT and Horn. The size of backdoors with respect to purely syntactically defined sub-solvers is relatively large. On the other hand, it is possible that a simplified sub-problem is polynomial-time solvable before an algorithmically defined sub-solver finds a solution. Therefore, we propose a sub-solver that first applies unit propagation, and then checks polynomial-time tractable classes.

We propose both systematic and local search algorithms for finding backdoors. The systematic search algorithms are guaranteed to find all minimal sized backdoors but are unable to handle large instances. Kilby, Slaney, Thiébaux, and Walsh [7] propose a local search algorithm to find small weak backdoors. Building on their work, we propose two local search algorithms for finding small backdoors. Our first algorithm incorporates our definition of sub-solver with Kilby et al.'s algorithm. Our second algorithm is a novel local search technique. We experiment on large real-world instances, including the instances from SAT-Race 2008, to compare our proposed algorithms to previous algorithms. Our algorithms based on our proposed sub-solvers can find smaller backdoors and significantly larger numbers of backdoors than previous algorithms.

## 2 Background

In this section, we review the necessary background in propositional satisfiability and backdoors in SAT instances.

We consider propositional formula in conjunctive normal form (CNF). A *literal* is a Boolean variable or its negation. A *clause* is a disjunction of literals. A clause with one literal is called a *unit clause* and the literal in the unit clause is called a *unit literal*. A propositional formula $F$ is in *conjunctive normal form* if it is a conjunction of clauses.

Given a propositional formula in CNF, the problem of determining whether there exists a variable assignment that makes the formula evaluate to *true* is called the *propositional satisfiability problem* or *SAT*. Propositional satisfiability is often solved using backtracking search. A backtracking search for a solution to a SAT instance can be seen as performing a depth-first traversal of a search tree. The search tree is generated as the search progresses and represents alternative choices that may have to be examined in order to find a solution or prove that no solution exists. Exploring a choice is also called *branching* and the order in which choices are explored is determined by a *variable ordering heuristic*. When specialized to SAT solving, backtracking algorithms are often referred to as being DPLL-based, in honor of Davis, Putnam, Logemann, and Loveland, the authors of one of the earliest works in the field [1].

Let $F$ denote a propositional formula. We use the value 0 interchangeably with *false* and the value 1 interchangeably with *true*. The notation $F[v = 0]$ represents a new formula, called the *residual formula*, obtained by removing all

clauses that contain the literal $\neg v$ and deleting the literal $v$ from all clauses. Similarly, the notation $F[v = 1]$ represents the residual formula obtained by removing all clauses that contain the literal $v$ and deleting the literal $\neg v$ from all clauses. Let $a_S$ be a set of assignments. The residual formula $F[a_S]$ is obtained by cumulatively reducing $F$ by each of the assignments in $a_S$.

*Example 1.* For example, the formula, $F = (x \vee \neg y) \wedge (x \vee y \vee z) \wedge (y \vee \neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u)$, is in CNF. Suppose $x$ is assigned *false*. The residual formula is given by, $F[x = 0] = (\neg y) \wedge (y \vee z) \wedge (y \vee \neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u)$.

As is clear, a CNF formula is satisfied only if each of its clauses is satisfied and a clause is satisfied only if at least one of its literals is equivalent to *true*. In a unit clause, there is no choice and the value of the literal is said to be *forced*. The process of *unit propagation* repeatedly assigns all unit literals the value *true* and simplifies the formula (i.e., the residual formula is obtained) until no unit clause remains or a conflict is detected. A conflict occurs when implications for setting the same variable to both *true* and *false* are produced.

*Example 2.* Consider again the formula $F[x = 0]$ given in Example 1. The unit clause $(\neg y)$ forces $y$ to be assigned 0. The residual formula is, $F[x = 0, y = 0] = (z) \wedge (\neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u)$. In turn, the unit clause $(z)$ forces $z$ to be assigned 1. Similarly, the assignments $w = 1$, $v = 1$, and $u = 1$ are forced.

Williams, Gomes, and Selman [13] formally define weak and strong backdoors. The definitions rely on the concept of a sub-solver $A$ that, given a formula $F$, in *polynomial time* either rejects the input or correctly solves $F$. A sub-solver can be defined either algorithmically or syntactically. For example, a DPLL-based SAT solver can be modified to be an algorithmically defined sub-solver by using just unit propagation, and returning "reject" if branching is required, "unsatisfiable" if a contradiction is encountered, and "satisfiable" if a solution is found. Examples of tractable syntactic classes include 2SAT, Horn, anti-Horn, and RHorn formulas. A formula is 2SAT if every clause contains at most two literals, Horn if every clause has at most one positive literal, anti-Horn if every clause has at most one negative literal, and renamable Horn (RHorn) if it can be transformed into Horn by a uniform renaming of variables.

A weak backdoor is a subset of variables such that some value assignment leads to a polynomial-time solvable sub-problem.

**Definition 1 (Weak Backdoor).** *A nonempty subset $S$ of the variables is a weak backdoor in $F$ for a sub-solver $A$ if there exists an assignment $a_S$ to the variables in $S$ such that $A$ returns a satisfying assignment of $F[a_S]$.*

*Example 3.* Consider once again the formula $F[x = 0]$ given in Example 2. After unit propagation every variable has been assigned a value and the formula $F$ is satisfied. Hence, $x$ is a weak backdoor in $F$ with respect to unit propagation.

A strong backdoor is a subset of variables such that every value assignment leads to a polynomial-time solvable sub-problem.

**Table 1.** Summary of previous experimental studies, where DPLL means the sub-solver used was defined algorithmically based on a DPLL solver; otherwise the sub-solver was defined syntactically using the given tractable class.

| | Sub-solvers | Instance domains |
|---|---|---|
| Williams et al. [13] | DPLL | structured |
| Interian [6] | 2SAT, Horn | random 3SAT |
| Dilkina et al. [2] | DPLL, Horn, RHorn | graph coloring, planning, game theory, automotive configuration |
| Paris et al. [9] | RHorn | random 3SAT, SAT competition |
| Kottler et al. [8] | 2SAT, Horn, RHorn | SAT competition, automotive configuration |
| Samer & Szeider [11] | Horn, RHorn | automotive configuration |
| Ruan et al. [10] | DPLL | quasigroup completion, graph coloring |
| Kilby et al. [7] | DPLL | random 3SAT |
| Gregory et al. [4] | DPLL | planning, graph coloring, quasigroup |
| Dilkina et al. [3] | DPLL | planning, circuits |

**Definition 2 (Strong Backdoor).** *A nonempty subset $S$ of the variables is a strong backdoor in $F$ for a sub-solver $A$ if for all assignments $a_S$ to the variables in $S$, $A$ returns a satisfying assignment or concludes unsatisfiability of $F[a_S]$.*

A *minimal backdoor* for an instance is a backdoor $S$ such that for every other backdoor $S'$, $|S| \leq |S'|$. A *small backdoor* refers to a backdoor $S$ such that no proper subset of $S$ is also a backdoor. A minimal backdoor can be viewed as a global minimum, and a small backdoor can be viewed as a local minimum.

## 3 Related Work

In this section, we review previous work on algorithms for finding weak and strong backdoors in SAT and their experimental evaluation (see Table 1).

In terms of algorithms, Williams, Gomes, and Selman [13, 14] present a systematic algorithm which searches every subset of variables for a backdoor. We modify this algorithm to find minimal backdoors. Interian [6] and Kilby, Slaney, Thiébaux, and Walsh [7] propose a local search algorithm for finding backdoors. Our algorithms build on Kilby et al's. We discuss this all in detail in Section 4.

In terms of experimental results, Dilkina, Gomes, and Sabharwal [2] show that strong Horn backdoors can be considerably larger than strong backdoors with respect to DPLL sub-solvers. Kottler, Kaufmann, and Sinz [8] compare 2SAT, Horn and RHorn and find that RHorn usually results in smaller backdoors. Samer and Szeider [11] compare Horn and RHorn strong backdoors and find as well that RHorn gives smaller backdoors. In general, previous experimental results show that backdoors with respect to syntactically defined sub-solvers have larger sizes than backdoors with respect to algorithmically defined sub-solvers.

In contrast to previous work, which consider DPLL, 2SAT, Horn, and RHorn sub-solvers independently, we combine syntactic and algorithmic sub-solvers into a single sub-solver. We also propose improved algorithms and experimentally evaluate our proposals on larger, more varied instances.

**Table 2.** Algorithms for finding weak and strong backdoors.

| | |
|---|---|
| EXACT | Our exact algorithm for finding minimal weak backdoors in satisfiable instances; |
| STRONG | Our exact algorithm for finding minimal strong backdoors in unsatisfiable instances; |
| KILBY | Kilby et al.'s [7] local search algorithm for finding small weak backdoors; |
| KILBYIMP | Kilby et al.'s [7] algorithm that incorporates our definition of sub-solver; |
| TABU | Our proposed local search algorithm for finding small weak backdoors. |

## 4 Algorithms for Finding Backdoors

In this section, we introduce how we define sub-solvers and describe several algorithms for finding backdoors (see Table 2).

In our proposed framework, we define the sub-solver both algorithmically and syntactically. Specifically, given a partial assignment to a subset of variables $S$, we first apply unit propagation and then check the following conditions to see if the resulting formula $F$ belongs to a polynomial-time tractable class:

1. if $F$ is satisfied;
2. if $F$ is 2SAT;
3. if $F$ is satisfied after assigning 0 (*false*) to every unassigned variable;
4. if $F$ is satisfied after assigning 1 (*true*) to every unassigned variable.

If one of the above conditions is true, then $S$ is a backdoor set. The first two conditions are trivial. The third condition covers (a superset of) Horn formula, while the last condition covers (a superset of) anti-Horn formula. If $F$ is a Horn formula, it can be satisfied by assigning 0 to all variables unless it has unit clauses with a single positive literal. However, after unit propagation, $F$ is guaranteed to have at least two unassigned literals in each clause. A similar reasoning applies if $F$ is anti-Horn. In specifying our algorithms, we make use of the following low level procedures, where $F$ is a CNF formula and $v$ is a Boolean value.

| | |
|---|---|
| `isSatisfied`($F$) | return true iff $F$ is already satisfied; |
| `is2SAT`($F$) | return true iff $F$ is a 2SAT formula; |
| `isSat2SAT`($F$) | return true iff $F$ is a satisfiable 2SAT formula; |
| `setVal`($F, v$) | return true iff $F$ is satisfied after assigning $v$ to every unassigned variable. |

### 4.1 Exact Algorithms: EXACT and STRONG

We describe exact algorithms, which are suitable for small instances with small backdoors. Algorithm EXACT takes as input a formula $F$ and finds all backdoors of size at most $k$ by performing an exhaustive search. We run algorithm EXACT with $k = 1, 2, \ldots, n-1$, until minimal backdoors are found.

The algorithm calls procedure `expand`($V, S, k$), which explores the variables of $F$ in a depth-first manner. Given a set of variables $V$, a set of minimal backdoors

$S$, and a positive integer $k$, the procedure returns true iff $V$ is a backdoor and as a side-effect, $S$ is updated. Given a value assignment to the variables in $V$, $V$ is a backdoor if there is no conflict after unit propagation and the resulting formula $F$ is in one of the polynomial-time tractable classes. The procedure recursively calls itself with one more variable added to $V$ and $k-1$.

**Algorithm:** EXACT$(F, k)$
$S \leftarrow \emptyset$;
**for** $i \leftarrow 0$ **to** $n-1$ **do** `expand(`$\{x_i\}$, $S$, $k$`)`;
**return** $S$;

**Procedure:** `expand`$(V, S, k)$
**foreach** *value assignment* $a_V$ *of* $V$ **do**
  **if** *unit propagation of* $a_V$ *does not result in conflicts* **then**
    **if** `isSatisfied(`$F$`)` $\vee$ `isSat2SAT(`$F$`)` $\vee$ `setVal(`$F$`, 0)` $\vee$ `setVal(`$F$`, 1)`
    **then** $S \leftarrow S \cup V$; **return** true;
**if** $k \leq 1$ **then return** false;
$j \leftarrow$ index of the last variable in $V$;
**for** $i \leftarrow (j+1)$ **to** $n-1$ **do** `expand(`$V \cup \{x_i\}$, $S$, $k-1$`)`;
**return** false;

The EXACT algorithm can easily be modified to give algorithm STRONG, which finds minimal strong backdoors in *unsatisfiable* instances. The idea is that if every value assignment to a set of variables $V$ results in conflicts during unit propagation, then we are able to conclude the unsatisfiability of the instance. Thus, $V$ is added to the list of strong backdoors.

### 4.2 Local Search Algorithms: KILBY, KILBYIMP, and TABU

The exact algorithms based on depth-first search are complete, but do not scale up to instances with larger backdoors. Here we discuss local search algorithms. In the local search algorithms each search state $s$ is a backdoor, and the cost of a node $s$ is the cardinality of the backdoor $s$.

Kilby et al. [7] propose algorithm KILBY for finding small weak backdoors using local search. Given a formula $F$, the DPLL solver Satz-rand is first used to solve $F$, recording the set $W$ of branching literals and the solution $M$. The set $W$ is an initial backdoor as Satz-rand is able to solve $F[W]$ without branching. Then, algorithm KILBY takes the inputs $F$, $W$, and $M$ to find small backdoors. The set $B$ is the current smallest backdoor. The algorithm has three constants: *RestartLimit*, which controls the number of restarts, a technique for escaping from local minima; *IterationLimit*, which controls the amount of search between restarts; and *CardMult*, which defines the neighbors of the current candidate backdoor $W$. In each iteration, the algorithm randomly selects from $M$ a set $Z$ of $|W| \times$ *CardMult* literals that are not in $W$. The set $Z$ of literals is appended to $W$, and procedure `minWeakBackdoor` is called to reduce the set $W \cup Z$ of literals into a small backdoor, which is the next search state.

**Algorithm:** KILBY$(F, W, M)$

$S \leftarrow \emptyset$, $B \leftarrow W$;
*RestartLimit* $\leftarrow 2$; *RestartCount* $\leftarrow 0$; *IterationLimit* $\leftarrow \sqrt{n} \times 3$; *CardMult* $\leftarrow 2$;
**while** *RestartCount* $<$ *RestartLimit* **do**
    *RestartCount* $\leftarrow$ *RestartCount* $+ 1$;
    $W \leftarrow B$;
    **for** $i \leftarrow 0$ **to** *IterationLimit* **do**
        $Z \leftarrow |W| \times$ *CardMult* literals chosen randomly from $M \setminus W$;
        $W \leftarrow$ `minWeakBackdoor(`$F$, $W \cup Z$`)`;
        **if** $|W| \leq |B|$ **then** $S \leftarrow S \cup W$;
        **if** $|W| < |B|$ **then** $B \leftarrow W$; *RestartCount* $\leftarrow 0$;
**return** $S$;

**Procedure:** `minWeakBackdoor`$(F, I)$

$W \leftarrow \emptyset$;
**while** $I \neq \emptyset$ **do**
    Choose literal $l \in I$; $I \leftarrow I \setminus \{l\}$;
    **if** DPLL *applied to* $F[W \cup I]$ *requires branching* **then**
        `// The following if statement is added in our sub-solver;`
1       **if** $\neg$`is2SAT(`$F$`)` $\wedge$ $\neg$`setVal(`$F, 0$`)` $\wedge$ $\neg$`setVal(`$F, 1$`)` **then**
            $W \leftarrow W \cup \{l\}$;
**return** $W$;

Kilby et al. use a simple sub-solver, which applies Satz-rand's unit propagation. We modify their algorithm KILBY to use the more sophisticated sub-solver we define. Algorithm KILBYIMP is the local search algorithm that results from adding Line 1 in procedure `minWeakBackdoor`. One further difference is that our DPLL solver is Minisat, where Minisat has a powerful pre-processor.

We also propose a novel algorithm TABU, which uses local search techniques, including Tabu Search, a best improvement strategy, and auxiliary local search. The search state $W$ is the current candidate backdoor, and *tabuList* is a list of previously visited search states. The tabu tenure is set to 30 to prevent our TABU from revisiting the last 30 search states. When the tabu list is full, the oldest state is replaced by the new state. The procedure `searchNeighbors`$(W, S, M)$ evaluates the neighborhood of $W$ and updates $W$ with the best improving neighbor not in *tabuList*. The **while** loop stops if no new small backdoors have been found in the last *RestartLimit* iterations. The procedure `localImprovement`$(S, M)$ is an auxiliary local search over the neighborhood of newly found small backdoors.

The procedure `searchNeighbors`$(W, S, M)$ explores all *IterationLimit* neighbors of the current backdoor $W$ to find a best non-tabu candidate backdoor. This is in contrast to Algorithm KILBY, which selects the first neighbor $s'$ encountered in the neighborhood of $s$ without considering the cost of $s'$; i.e., $|s'|$. The value of *minCost* is the minimal size of backdoors in *Neighbor*. If *minCost* is no larger than the size of the current smallest backdoor, then all the backdoors in *Neighbor* of size *minCost* are added to the list of small backdoors $S$. A small backdoor of size *minCost* is randomly selected from *Neighbor* to be the next search state.

When *minCost* is larger than the size of the current smallest backdoor, the search can escape from local minima by making worse moves. If every non-tabu candidate backdoor in *Neighbor* has a larger size than the current smallest backdoor, the search moves to a best candidate backdoor from *Neighbor*.

**Algorithm:** TABU$(F, W, M)$

$W \leftarrow$ `minWeakBackdoor(`$F$, $W$`)`;
$preSize \leftarrow |S|$; $RestartLimit \leftarrow 2$; $RestartCount \leftarrow 0$; $tabuList \leftarrow \emptyset$;
**while** $RestartCount < RestartLimit$ **do**
  $RestartCount \leftarrow RestartCount + 1$;
  $cost \leftarrow$ `searchNeighbors(`$W$, $S$, $M$`)`;
  **if** $cost = 0$ **then** break;
  $tabuList \leftarrow tabuList \cup W$;
  **if** $|S| > preSize$ **then** $RestartCount \leftarrow 0$;
  $preSize \leftarrow |S|$;
$tabuList \leftarrow \emptyset$;
`localImprovement(`$S$, $M$`)`;
**return** $S$;

**Procedure:** `searchNeighbors`$(W, S, M)$

$IterationLimit \leftarrow \sqrt{n} \times 2$; $CardMult \leftarrow 2$; $Neighbor \leftarrow \emptyset$, $Cost \leftarrow \emptyset$;
**for** $i \leftarrow 0$ **to** $IterationLimit$ **do**
  $Z \leftarrow |W| \times CardMult$ literals chosen randomly from $M \setminus W$;
  $W \leftarrow$ `minWeakBackdoor(`$F$, $W \cup Z$`)`;
  **if** $W \notin tabuList$ **then**
    $Neighbor \leftarrow Neighbor \cup W$;
    $Cost \leftarrow Cost \cup |W|$;
**if** $|Neighbor| = 0$ **then return** 0;
$minCost \leftarrow \min(Cost)$;
**if** $minCost \leq$ *current smallest backdoor size* **then**
  $S \leftarrow S \cup \{B \in Neighbor \mid |B| = minCost\}$;
$W \leftarrow$ select a backdoor from *Neighbor* with size *minCost* randomly;
**return** *minCost*;

The procedure `localImprovement`$(S, M)$ is an auxiliary local search that attempts to find more minimal backdoors by replacing variables in $s$. The inspiration for the procedure is the observation that some variables appear in most backdoors and some backdoor sets only differ from each other by one variable.

**Procedure:** `localImprovements`$(S, M)$

**foreach** *new backdoor* $B \in S$, $B \notin tabuList$ **do**
  $tabuList \leftarrow tabuList \cup B$;
  **foreach** *literal* $l \in \{M \setminus B\}$ **do**
    $B \leftarrow$ `minWeakBackdoor(`$F$, $B \cup l$`)`;
    **if** $|B| \leq$ *current minimum backdoor size* **then** $S \leftarrow S \cup B$;

**Table 3.** Size, percentage, and number of minimal backdoors found by the EXACT algorithm when applied to small real-world instances with $n$ variables and $m$ clauses.

| Instance | $n$ | $m$ | BD size (%) | # BDs |
|---|---|---|---|---|
| grieu-vmpc-s05-24s | 576 | 49478 | 3 (0.52%) | 143 |
| een-tip-sat-texas-tp-5e | 17985 | 153 | 1 (0.01%) | 2 |
| anomaly | 48 | 182 | 1 (2.08%) | 2 |
| medium | 116 | 661 | 1 (0.86%) | 5 |
| huge | 459 | 4598 | 2 (0.44%) | 89 |
| bw_large.a | 459 | 4598 | 2 (0.44%) | 89 |
| bw_large.b | 1087 | 13652 | 2 (0.18%) | 7 |

## 5 Experimental Evaluation

In this section, we describe experiments on structured and real-world SAT instances to compare the algorithms shown in Table 2. The set of satisfiable test instances consists of planning instances from SATLIB [5] and all but six of the satisfiable real-world instances from SAT-Race 2008 (the instances excluded were those that Minisat was unable to solve within the competition time limit). The set of unsatisfiable test instances is from the domain of automotive configuration [12]. The instances were all pre-processed with Minisat, which can sometimes greatly reduce the number of clauses. The experiments were run on the Whale cluster of the SHARCNET system (www.sharcnet.ca). Each node of the cluster is equipped with four Opteron CPUs at 2.2 GHz and 4.0 GB memory.

### 5.1 Experiments on Finding Weak Backdoors

Algorithm EXACT is able to find all minimal backdoors for instances with small backdoors (see Table 3). The sizes of minimal backdoors in the blocks world instances are smaller than those reported by Dilkina et al. [3] who report percentages between 1.09% to 4.17% even though they used clause learning in addition to unit propagation. The reason is that our sub-solver not only applies unit propagation, but also tests for polynomial-time syntactic classes. Systematic algorithms do not scale up to instances with larger backdoors, though.

We also compared the small backdoors found by the local search algorithms, KILBY, KILBYIMP, and TABU. With different initial solutions as inputs, the local search algorithms were run repeatedly until a cutoff time was reached. Only the smallest backdoors found by the algorithms were recorded. The cutoff time was set to 3 hours for instances with fewer than 10,000 variables (see Table 4) and 15 hours for larger instances (see Table 5). For each instance, the algorithm that found the smallest backdoors among the three local search algorithms is highlighted, with the largest number of backdoors used to break ties.

When the cutoff time was reached, we waited for the algorithms to finish the current iteration. Because TABU takes longer to complete one iteration than KILBY and KILBYIMP, the time when TABU found small backdoors in some SAT-Race 2008 instances was a little longer than 15 hours. The longest time recorded

**Table 4.** Size, percentage, and number of small backdoors found by the local search algorithms within a cutoff of 3 hours when applied to real-world instances with $n$ variables ($n < 10,000$) and $m$ clauses.

| | | | Kilby | | KilbyImp | | Tabu | |
|---|---|---|---|---|---|---|---|---|
| Instance | $n$ | $m$ | BD size (%) | # BDs | BD size (%) | # BDs | BD size (%) | # BDs |
| SAT Competition 2002 | | | | | | | | |
| apex7_gr_rcs_w5.shuffled | 1500 | 11136 | 77 (5.13%) | 1 | **47 (3.13%)** | **4** | 53 (3.53%) | 42885 |
| dp10s10.shuffled | 8372 | 8557 | 9 (0.11%) | 10520 | 9 (0.11%) | 9573 | **9 (0.11%)** | **59399** |
| bart11.shuffled | 162 | 675 | 15 (9.26%) | 4190 | 14 (8.64%) | 2903 | **14 (8.64%)** | **45044** |
| SAT-Race 2005 and 2008 | | | | | | | | |
| grieu-vmpc-s05-24s | 576 | 49478 | 3 (0.52%) | 143 | 3 (0.52%) | 143 | 3 (0.52%) | 143 |
| grieu-vmpc-s05-27r | 729 | 71380 | 4 (0.55%) | 710 | 4 (0.55%) | 660 | **4 (0.55%)** | **3271** |
| simon-mixed-s02bis-01 | 2424 | 13793 | 8 (0.33%) | 566 | 8 (0.33%) | 566 | **8 (0.33%)** | **10440** |
| simon-s02b-r4b1k1.2 | 2424 | 13811 | 8 (0.33%) | 394 | 7 (0.29%) | 3 | **7 (0.29%)** | **16** |
| Blocks world planning | | | | | | | | |
| bw_large.c | 3016 | 50237 | 4 (0.13%) | 1934 | **3 (0.10%)** | **15** | **3 (0.10%)** | **15** |
| bw_large.d | 6325 | 131607 | 6 (0.10%) | 790 | **5 (0.08%)** | **69** | 6 (0.10%) | 640 |
| Logistics planning | | | | | | | | |
| logistics.a | 828 | 3116 | 20 (2.42%) | 147 | **20 (2.42%)** | **6675** | 24 (2.90%) | 584257 |
| logistics.b | 843 | 3480 | 16 (1.90%) | 1688 | **15 (1.78%)** | **9789** | 16 (1.90%) | 7634 |
| logistics.c | 1141 | 5867 | 26 (2.28%) | 18 | **25 (2.19%)** | **387** | 28 (2.45%) | 424467 |
| logistics.d | 4713 | 16588 | 25 (0.53%) | 39 | **22 (0.47%)** | **61** | 28 (0.59%) | 36610 |

was 168 seconds after the 15-hour cutoff time. It is possible that Kilby and KilbyImp would have found smaller backdoors during this leeway. Although Tabu takes longer in one iteration than Kilby and KilbyImp, Tabu is sometimes able to find a larger number of backdoors in the given time, and for instances that have small backdoors of size less than 10, a remarkably larger number. For many more of these real-world instances, KilbyImp outperformed Kilby and Tabu in finding small backdoors. Both Kilby and KilbyImp select the first candidate backdoor encountered. The Tabu algorithm searches the entire neighborhood for the best improvement, which can be too expensive when the backdoor size and the total number of variables are large.

Williams et al. [13] experimented on practical instances with fewer than 10,000 variables and showed that such instances had relatively small backdoors. We extend their result to the SAT-Race 2008 instances, which have a huge number of variables and clauses. The SAT-Race 2008 instances have backdoors that consist of hundreds of variables. However, the backdoor size is usually less than 0.5% of the total number of variables. Thus, our results agree with Williams et al. that practical instances generally have small tractable structures.

### 5.2 Experiments on Finding Strong Backdoors

In previous work [11, 2], unsatisfiable SAT benchmarks from automotive configuration [12] were used in the experiments. Among the 84 unsatisfiable instances, Minisat concludes the unsatisfiability of 71 instances after pre-processing. We applied the Strong algorithm to find minimal strong backdoors for the remaining 13 instances (see Table 6). The sizes of minimal strong backdoors range from

**Table 5.** Size, percentage, and number of small backdoors found by the local search algorithms within a cutoff of 15 hours when applied to real-world instances with $n$ variables ($n > 10,000$) and $m$ clauses. An entry of *timeout* indicates that the local search algorithm failed to find any small backdoor within the cutoff time.

| | | | Kilby | | KilbyImp | | Tabu | |
|---|---|---|---|---|---|---|---|---|
| Instance | $n$ | $m$ | BD size (%) | # BDs | BD size (%) | # BDs | BD size (%) | # BDs |
| ibm-2002-04r-k80 | 104450 | 238773 | 252 (0.24%) | 10 | **154 (0.15%)** | **53** | 184 (0.18%) | 2 |
| ibm-2002-11r1-k45 | 156626 | 290625 | 307 (0.20%) | 3 | **282 (0.18%)** | **7** | 344 (0.22%) | 2 |
| ibm-2002-18r-k90 | 175216 | 370661 | 360 (0.21%) | 3 | **331 (0.19%)** | **6** | 496 (0.28%) | 1 |
| ibm-2002-20r-k75 | 151202 | 319192 | 319 (0.21%) | 4 | **275 (0.18%)** | **17** | 384 (0.25%) | 1 |
| ibm-2002-22r-k75 | 191166 | 399095 | 453 (0.24%) | 4 | **424 (0.22%)** | **3** | 551 (0.29%) | 2 |
| ibm-2002-22r-k80 | 203961 | 427792 | 499 (0.25%) | 1 | **466 (0.23%)** | **4** | 605 (0.30%) | 1 |
| ibm-2002-23r-k90 | 222291 | 469900 | 537 (0.24%) | 2 | **534 (0.24%)** | **1** | 624 (0.28%) | 2 |
| ibm-2002-29r-k75 | 64686 | 258748 | 81 (0.13%) | 11 | **58 (0.09%)** | **26** | 59 (0.09%) | 1 |
| ibm-2004-01-k90 | 64699 | 201260 | 148 (0.23%) | 2 | **87 (0.13%)** | **5** | 93 (0.14%) | 8 |
| ibm-2004-1_11-k80 | 262808 | 565220 | 696 (0.27%) | 4 | **648 (0.25%)** | **1** | 732 (0.28%) | 1 |
| ibm-2004-23-k100 | 207606 | 481764 | 524 (0.25%) | 2 | **455 (0.22%)** | **1** | 618 (0.30%) | 4 |
| ibm-2004-23-k80 | 165606 | 379170 | 465 (0.28%) | 2 | **441 (0.27%)** | **1** | 550 (0.33%) | 1 |
| ibm-2004-29-k55 | 37714 | 123699 | 67 (0.18%) | 16 | 52 (0.14%) | 21 | **49 (0.13%)** | **6381** |
| ibm-2004-3_02_3-k95 | 73525 | 169473 | 1297 (1.76%) | 1 | **238 (0.32%)** | **2** | 251 (0.34%) | 1 |
| mizh-md5-47-3 | 65604 | 153650 | **179 (0.27%)** | **1** | **179 (0.27%)** | **1** | 265 (0.40%) | 4 |
| mizh-md5-47-4 | 65604 | 153778 | **184 (0.28%)** | **2** | 190 (0.29%) | 1 | 232 (0.35%) | 2 |
| mizh-md5-47-5 | 65604 | 153896 | **181 (0.28%)** | **2** | **181 (0.28%)** | **2** | 235 (0.36%) | 1 |
| mizh-md5-48-2 | 66892 | 157184 | **203 (0.30%)** | **1** | **203 (0.30%)** | **1** | 289 (0.43%) | 1 |
| mizh-md5-48-5 | 66892 | 157466 | **189 (0.28%)** | **6** | **189 (0.28%)** | **6** | 238 (0.36%) | 1 |
| mizh-sha0-35-3 | 48689 | 115548 | 258 (0.53%) | 1 | 254 (0.52%) | 2 | **238 (0.49%)** | **1** |
| mizh-sha0-35-4 | 48689 | 115631 | 237 (0.49%) | 1 | 237 (0.49%) | 1 | **210 (0.43%)** | **1** |
| mizh-sha0-36-1 | 50073 | 120102 | 261 (0.52%) | 1 | 261 (0.52%) | 1 | **219 (0.44%)** | **1** |
| mizh-sha0-36-3 | 50073 | 120212 | 249 (0.50%) | 1 | 260 (0.52%) | 4 | **209 (0.42%)** | **5** |
| mizh-sha0-36-4 | 50073 | 120279 | 237 (0.47%) | 1 | 237 (0.47%) | 1 | **220 (0.44%)** | **1** |
| post-c32s-gcdm16-22 | 129652 | 88631 | 12 (0.01%) | 133 | 12 (0.01%) | 133 | **11 (0.01%)** | **126** |
| velev-fvp-sat-3.0-b18 | 35853 | 968394 | 228 (0.64%) | 3 | **212 (0.59%)** | **1** | 227 (0.63%) | 1 |
| velev-vliw-sat-4.0-b4 | 520721 | 13348080 | timeout | | timeout | | **933 (0.18%)** | **1** |
| velev-vliw-sat-4.0-b8 | 521179 | 13378580 | timeout | | timeout | | timeout | |
| een-tip-sat-nusmv-t5.B | 61933 | 42043 | 109 (0.18%) | 6 | **88 (0.14%)** | **35** | 92 (0.15%) | 14318 |
| een-tip-sat-vis-eisen | 18607 | 12801 | 8 (0.04%) | 6087 | 8 (0.04%) | 16466 | **8 (0.04%)** | **36941** |
| narain-vpn-clauses-8 | 1461772 | 4572347 | timeout | | timeout | | timeout | |
| palac-sn7-ipc5-h16 | 114548 | 218043 | 10 (0.01%) | 46 | 10 (0.01%) | 46 | **10 (0.01%)** | **1533** |
| palac-uts-l06-ipc5-h34 | 187667 | 606674 | **10 (0.01%)** | **152** | **10 (0.01%)** | **152** | 10 (0.01%) | 102 |
| schup-l2s-motst-2-k315 | 507145 | 590065 | timeout | | timeout | | timeout | |
| simon-s03-w08-15 | 132555 | 269328 | 233 (0.18%) | 26 | **115 (0.09%)** | **31** | 152 (0.12%) | 4 |

1 to 3, which are smaller than the sizes reported in [11, 2]. We found smaller backdoors because we applied a systematic search algorithm, and we defined sub-solvers both syntactically and algorithmically.

## 6 Conclusion

We presented exact algorithms for finding all minimal weak backdoors in satisfiable instances and all minimal strong backdoors in unsatisfiable instances. Building on Kilby et al.'s local search algorithm Kilby, we described our improved local search algorithms KilbyImp and Tabu for finding small weak backdoors. We empirically evaluated the algorithms on structured and real-world SAT instances. The experimental results show that our algorithms based on our proposed sub-solvers can find smaller backdoors and significantly larger numbers

**Table 6.** Size and number of minimal strong backdoors found by the STRONG algorithm when applied to automotive configuration instances with $n$ variables and $m$ clauses.

| Instance | $n$ | $m$ | BD size | BD # | Instance | $n$ | $m$ | BD size | BD # |
|---|---|---|---|---|---|---|---|---|---|
| C168_FW_SZ_128 | 1698 | 5425 | 3 | 6 | C168_FW_SZ_66 | 1698 | 5401 | 1 | 3 |
| C202_FS_RZ_44 | 1750 | 6199 | 2 | 26 | C202_FW_SZ_87 | 1799 | 8946 | 3 | 90 |
| C210_FS_RZ_23 | 1755 | 5778 | 3 | 17 | C210_FS_RZ_38 | 1755 | 5763 | 2 | 4 |
| C210_FS_SZ_103 | 1755 | 5775 | 2 | 3 | C210_FW_RZ_30 | 1789 | 7426 | 3 | 16 |
| C210_FW_RZ_57 | 1789 | 7405 | 2 | 4 | C210_FW_SZ_106 | 1789 | 7417 | 2 | 3 |
| C210_FW_SZ_128 | 1789 | 7412 | 1 | 3 | C210_FW_UT_8630 | 2024 | 9721 | 1 | 2 |
| C220_FV_SZ_65 | 1728 | 4496 | 1 | 2 | | | | | |

of backdoors than previous algorithms. In future work, we intend to use our algorithms for finding backdoors to study value and variable ordering mistakes and their effect on the performance of backtracking algorithms.

## References


1. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Commun. ACM*, 5(7):394–397, 1962.
2. B. Dilkina, C. P. Gomes, and A. Sabharwal. Tradeoffs in the complexity of backdoor detection. In: C. Bessiere (ed.) CP 2007. LNCS, vol. 4741, pp. 256–270. Springer, Heidelberg, 2007.
3. B. Dilkina, C. P. Gomes, and A. Sabharwal. Backdoors in the context of learning. In: O. Kullmann (ed.) SAT 2009. LNCS, vol. 5584, pp. 73–79. Springer, Heidelberg, 2009.
4. P. Gregory, M. Fox, and D. Long. A new empirical study of weak backdoors. In: P. J. Stuckey (ed.) CP 2008. LNCS, vol. 5202, pp. 618–623. Springer, Heidelberg, 2008.
5. H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In: I.P. Gent, H.v. Maaren, T. Walsh (eds.) SAT 2000, pp. 283–292, IOS Press, 2000.
6. Y. Interian. Backdoor sets for random 3-SAT. Paper presented at *SAT-2003*, 2003.
7. P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In: *Proc. of AAAI*, pp. 1368–1373, 2005.
8. S. Kottler, M. Kaufmann, and C. Sinz. Computation of renamable Horn backdoors. In: H. K. Büning and X. Zhao (eds.) SAT 2008. LNCS, vol. 4996, pp. 154–160. Springer, Heidelberg, 2008.
9. L. Paris, R. Ostrowski, P. Siegel, and L. Sais. Computing Horn strong backdoor sets thanks to local search. In: *Proc. of ICTAI*, pp. 139–143, 2006.
10. Y. Ruan, H. Kautz, and E. Horvitz. The backdoor key: A path to understanding problem hardness. In: *Proc. of AAAI*, pp. 124–130, 2004.
11. M. Samer and S. Szeider. Backdoor trees. In: *Proc. of AAAI*, pp. 363–368, 2008.
12. C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97, 2003.
13. R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In: *Proc. of IJCAI*, pp. 1173–1178, 2003.
14. R. Williams, C. Gomes, and B. Selman. On the connections between backdoors and heavy-tails on combinatorial search. Paper presented at *SAT-2003*, 2003.