algorithms

MDPI

Article

Boosting the Performance of CDCL-Based SAT Solvers by Exploiting Backbones and Backdoors

Tasniem Al-Yahya *, Mohamed El Bachir Abdelkrim Menai and Hassan Mathkour

Department of Computer Science, College of Computer and Information Sciences, King Saud University, P.O. Box 51178, Riyadh 11543, Saudi Arabia

* Correspondence: tnalyahya@imamu.edu.sa

Abstract: Boolean structural measures were introduced to explain the high performance of conflict-driven clause-learning (CDCL) SAT solvers on industrial SAT instances. Those considered in this study include measures related to backbones and backdoors: backbone size, backbone frequency, and backdoor size. A key area of research is to improve the performance of CDCL SAT solvers by exploiting these measures. For the purpose of guiding the CDCL SAT solver for branching on backbone and backdoor variables, this study proposes low-overhead heuristics for computing these variables. Through these heuristics, a set of modifications to the Variable State Independent Decaying Sum (VSIDS) decision heuristic is suggested to exploit backbones and backdoors and potentially improve the performance of CDCL SAT solvers. In total, fifteen variants of two competitive base solvers, MapleLCMDistChronoBT-DL-v3 and LSTech, were developed. Empirical evaluation was conducted on 32 industrial families from 2002–2021 SAT competitions. According to the results, modifying the VSIDS heuristic in the base solvers to exploit backbones and backdoors improves its performance. In particular, our new CDCL SAT solver, LSTech_BBsfcr_v1, solved more industrial SAT instances than the winning CDCL SAT solvers in 2020 and 2021 SAT competitions.

Keywords: SAT; CDCL; VSIDS; Boolean structural measures; backbone; backdoor

Citation: Al-Yahya, T.; Menai, M.E.B.A.; Mathkour, H. Boosting the Performance of CDCL-Based SAT Solvers by Exploiting Backbones and Backdoors. *Algorithms* **2022**, *15*, 302. https://doi.org/10.3390/a15090302

Academic Editor: Jan Friso Groote

Received: 24 July 2022
Accepted: 22 August 2022
Published: 26 August 2022

1. Introduction

The Boolean satisfiability problem (SAT) [1] is a fundamental NP-complete problem in automated reasoning and mathematical logic. As NP-complete problems can be reduced to SAT in polynomial time, SAT is applicable to a wide range of fields [2–4].

The annual SAT competitions have become an essential event for the distribution of SAT benchmarks and the development of new SAT-solving methods [5]. Sequential SAT solvers compete mainly in three categories: industrial, crafted, and random tracks. The SAT competitions have demonstrated how difficult it is for SAT solvers to perform well across all categories. Results show that conflict-driven clause-learning (CDCL) SAT solvers were most performant for solving industrial and crafted SAT benchmarks, whereas look-ahead and Stochastic Local Search (SLS)-based SAT solvers have dominated the random category [5]. Modern implementations of CDCL SAT solvers employ a lot of heuristics. Some of them can be considered baseline, such as the Variable State Independent Decaying Sum (VSIDS) [6], restarts [7], and Literal Block Distance (LBD) [8]. Several others were incorporated recently, including: Learnt Clause Minimization (LCM) [9], Distance (Dist) heuristic [10], Chronological Backtracking (ChronoBT) [11], duplicate learnts heuristic [12], Conflict History-Based (CHB) heuristic [13], Learning Rate-based Branching (LRB) heuristic [14], and the SLS component [15]. The results of the SAT competitions have led researchers to conclude that (1) industrial, crafted, and random SAT instances have distinct structures, and (2) SAT-solving methods could exploit such structures.

Boolean structural measure proposed for SAT include the phase transition [16], backbone size [17], and backdoor size [18,19]. We propose three new related measures to the

backbones and backdoors: the backbone frequency, backbone coverage, and backdoor coverage (The reader is referred to Appendix A where the evidence of the backbone and backdoor-related measures on industrial, crafted, and random benchmark instances drawn from 2002–2020 SAT competitions are investigated. In particular, we evaluated the backbone size, LSR backdoor size, and backbone/backdoor variable overlap size, along with the three proposed new related measures: backbone frequency, backbone coverage, and backdoor coverage.). Empirical results indicate that most random benchmarks have no backbones, whereas on average, industrial and crafted benchmark instances have small backbone sizes [20,21]. The frequency of backbones is low for all benchmark categories. As for the backbone coverage, industrial and crafted benchmarks have higher coverages, on average, than random ones. In both crafted and random benchmark instances, the backdoor size and coverage are greater than those in the industrial category [20,21]. Additionally, across all SAT benchmarks, there tends to be a little overlap between backbone and backdoor variables [20,21].

A fundamental problem of interest and practical importance concerns the possibility of enhancing the performance of SAT solvers by exploiting the inherent structure of the instances. Several studies have shown, for example, that backbone/backdoor-guided branching heuristics improve the solver performance [18,22–24]. Despite these efforts, this problem remains an open issue [19,25]. One reason is that the computational estimation of most Boolean structural measures (e.g., backbone and backdoor) is intractable [26]. Second, as a consequence, the structural measures of SAT and their impact on the behavior of SAT solvers have not been fully investigated [19,25].

The present study examines the relevance of Boolean structural measures on the performance of CDCL SAT solvers. The aim is to improve the performance of CDCL SAT solves by exploiting Boolean structural measures. In achieving this, low-overhead heuristics for computing backbones and backdoors are proposed. Using these heuristics, the VSIDS heuristic is extended to exploit backbones and backdoors. In particular, upon restart, conflict, and/or backtrack, variables that are likely to be backbones and/or backdoors are bumped. Accordingly, competitive CDCL SAT solver(s) are developed by improving two state-of-the-art CDCL SAT solvers, MapleLCMDistChronoBT-DL-v3 and LSTech, to exploit backbones and backdoors. The main contributions of this work are threefold:

1. To compute backbones and backdoors based on a low-overhead computational heuristic, in order to guide the CDCL SAT solver to branch on backbone/backdoor variables (Section 4.1).
2. To extend the VSIDS variable decision heuristic to exploit backbones and backdoors (Section 4.2).
3. To develop a competitive CDCL based SAT solver by improving two state-of-the-art solvers, MapleLCMDistChronoBT-DL-v3 and LSTech, to exploit backbones and backdoors (Sections 4 and 5).

The rest of the paper is organized as follows: In Section 2, the definitions and notations used throughout the paper are introduced. In Section 3, a review of state-of-the-art CDCL SAT solvers is provided. In particular, the base solvers that are the focus of this study, MapleLCMDistChronoBT-DL-v3 and LSTech, are described. In addition, algorithms for computing backbone and backdoor variables are reviewed. In Section 4, a description of the proposed CDCL SAT solvers that exploit backbones and backdoors is presented. In Section 5, results of the experimental evaluation are reported. This is followed, in Section 6, by a discussion of the main findings and issues. Finally, Section 7 contains concluding remarks and suggests future directions.

2. Definitions and Notations

The Boolean satisfiability problem is defined in this section, which is followed by formal definitions of backbones and backdoors.

2.1. Boolean Satisfiability Problem

A Boolean variable x may take the value $true$ or $false$. A literal is a Boolean variable x or its negation $\overline{x}$. A clause C is a disjunction of literals. Given a CNF formula ϕ defined over X, where $X = \{x_1, x_2, \ldots, x_n\}$ is a set of Boolean variables and $L = \{x_i, \overline{x}_i | x_i \in X, 1 \leq i \leq n\}$ is a set of literals over X, the following definitions hold:

Definition 1 (Conjunctive Normal Form (CNF) formula)**.** *A CNF formula ϕ is a conjunction of clauses, which is represented as a multiset of clauses $\{C_1, C_2, \ldots, C_m\}$.*

Definition 2 (k-CNF formula)**.** *A k-CNF formula is a CNF formula with exactly k literals per clause.*

Definition 3 (Assignment)**.** *An assignment is a mapping from each variable x_i to $\{true, false\}$. This is denoted by:*

$$\phi|_{\{x_i = value_i | \forall x_i \in X, value_i \in \{true, false\}\}}$$

Definition 4. *An assignment satisfies a literal x_i if $x_i = true$, and satisfies a literal $\overline{x}_i$ if $x_i = false$.*

Definition 5. *An assignment satisfies a clause C_i if it satisfies at least one literal in C_i.*

Definition 6. *An assignment satisfies a CNF formula if it satisfies all clauses.*

Definition 7. *A CNF formula ϕ is satisfiable if there exists an assignment that satisfies ϕ, that is:*

$$\phi|_{\{x_i = value_i | \forall x_i \in X, value_i \in \{true, false\}\}} = true$$

Otherwise, the formula ϕ is unsatisfiable:

$$\phi|_{\{x_i = value_i | \forall x_i \in X, value_i \in \{true, false\}\}} = false$$

Definition 8 (SAT problem [1])**.** *Given a CNF formula ϕ over X, the SAT problem asks whether there is an assignment that satisfies ϕ, or it decides that the formula is unsatisfiable.*

Definition 9 (SAT solver)**.** *A SAT solver is a computer program which aims to solve the SAT problem.*

2.2. Boolean Structural Measures

Definition 10 (Backbone literal [17])**.** *x_b is a backbone literal of ϕ if for all satisfying assignments of ϕ, the value of x_b is fixed.*

Definition 11 (Backbone of a CNF formula [17])**.** *The backbone of ϕ is the set of all backbone literals, which is denoted BB_ϕ.*

Definition 12 (Backbone size of a CNF formula [17])**.** *The backbone size of ϕ is the cardinality of BB_ϕ, which is denoted $|BB_\phi|$. The normalized backbone size of ϕ is the ratio of the backbone size to the number of variables $|X|$, which is denoted $BBsize_\phi$:*

$$BBsize_\phi = \frac{|BB_\phi|}{|X|} \tag{1}$$

Definition 13 (Backbone frequency)**.** *The backbone frequency of a backbone literal $x_b \in BB_\phi$ is the ratio of the total number of occurrences of x_b in ϕ to the total number of clauses (after CNF formula simplification, which involves reducing the number of variables and/or clauses in a CNF formula to a logically equivalent formula. It is part of the preprocessing phase that takes place before the solving phase.), which is denoted $BBfre_{x_b}$. This can be expressed as follows:*

$$BBfre_{x_b} = \frac{\sum_{\substack{i=1 \\ x_b \in C_i, C_i \in C}}^{|C|} 1}{|C|} \tag{2}$$

The backbone frequency of ϕ is the percentage average of the backbone frequencies of all backbone literals, which is denoted $BBfre_\phi$. This is expressed as follows:

$$BBfre_\phi = \frac{\sum_{\substack{b=1 \\ x_b \in BB_\phi}}^{|BB_\phi|} BBfre_{x_b}}{|BB_\phi|} \times 100 \tag{3}$$

Definition 14 (SAT sub-solver [18]). *A SAT sub-solver Γ is an algorithm that takes an input CNF formula ϕ and satisfies the following:*

1. *(Trichotomy) Γ correctly determines ϕ. If satisfiable it returns a solution; otherwise, it is unsatisfiable.*
2. *(Efficiency) Γ runs in polynomial time.*
3. *(Trivial-solvability) Γ can determine whether ϕ is trivially*
   - *true, that is, ϕ has no clauses; or*
   - *false, that is, ϕ has an empty clause.*
4. *(Self-reducibility) If Γ determines ϕ, then $\forall x_i \in X$, Γ determines:*

$$\phi^*|_{\{x_i = value_i | value_i \in \{true, false\}\}}. \tag{4}$$

where the latter denotes a simplified formula by fixing the value of any x_i to true or false.

Definition 15 (Weak backdoor [18]). *A non-empty subset of variables $BD_\phi \subseteq X$ is a weak backdoor with respect to sub-solver Γ if there exists a truth assignment of BD_ϕ s.t. Γ returns a satisfying assignment.*

Definition 16 (Strong backdoor [18]). *A subset of variables $BD_\phi \subseteq X$ is a strong backdoor with respect to sub-solver Γ if for all truth assignments of BD_ϕ, Γ returns a satisfying assignment or concludes unsatisfiability.*

Definition 17 (Learning-Sensitive (LS) backdoor [27]). *A subset of variables B_{LS} is an LS backdoor with respect to sub-solver Γ if there exists a search tree exploration order such that a CDCL SAT solver branches only on variables in B_{LS}, and with Γ and learnt clauses at the leaves of the search tree, it either finds a satisfying assignment for ϕ, or it proves that ϕ is unsatisfiable.*

Definition 18 (Learning Sensitive with Restarts (LSR) backdoor [19]). *A subset of variables B_{LSR} is an LSR backdoor with respect to sub-solver Γ if there exists a search tree exploration order with restarts such that a CDCL SAT solver branches only on variables in B_{LSR}, and with a sub-solver Γ and learnt clauses at the leaves of the search tree, it either finds a satisfying assignment for ϕ or proves that ϕ is unsatisfiable.*

Definition 19 (Backdoor size of a CNF formula [18]). *The backdoor size of ϕ is the cardinality of BD_ϕ, denoted by $|BD_\phi|$. The normalized backdoor size of a CNF formula ϕ is the ratio of the backdoor size to the number of variables of $|X|$, which is denoted $BDsize_\phi$. This can be expressed as follows:*

$$BDsize_\phi = \frac{|BD_\phi|}{|X|} \tag{5}$$

Definition 20 (Backbone/backdoor variable overlap [20]). *(To ensure the correctness of computing the backbone/backdoor variable overlap set, we relax the set BB_ϕ to be the set of variables,*

not literals. As such, for all backbone literals in BB_ϕ, their corresponding variables are considered.) Let BB_ϕ and BD_ϕ be the backbone and backdoor of a CNF formula ϕ, respectively. The backbone/backdoor variable overlap is referred to as BBD_ϕ, and it is given by:

$$BBD_\phi = BB_\phi \cap BD_\phi \tag{6}$$

The backbone/backdoor variable overlap size of a CNF formula ϕ is the cardinality of BBD_ϕ, which is denoted by $|BBD_\phi|$. The normalized backbone/backdoor size is the ratio of $|BBD_\phi|$ to the number of variables $|X|$, which is denoted by $BBDoverlap_\phi$:

$$BBDoverlap_\phi = \frac{|BBD_\phi|}{|X|} \tag{7}$$

3. Related Work

CDCL SAT solvers follow primarily the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [28], and they incorporate a number of effective techniques, including clause learning [29], lazy data structures [6], deletion policies [30], and restarts [31]. Typically, a CDCL SAT solver is organized into three main modules [6,29,32]: the decision module, used for branching; the deduction module, used for unit propagation and the identification of unsatisfied clauses (or conflicts); and the diagnosis module, which is used for conflict analysis and clause learning (see Figure 1). Accordingly, the main runtime breakdown of a CDCL SAT solver is: 10% decisions, 80% propagation, and 10% account for conflict analysis [6].

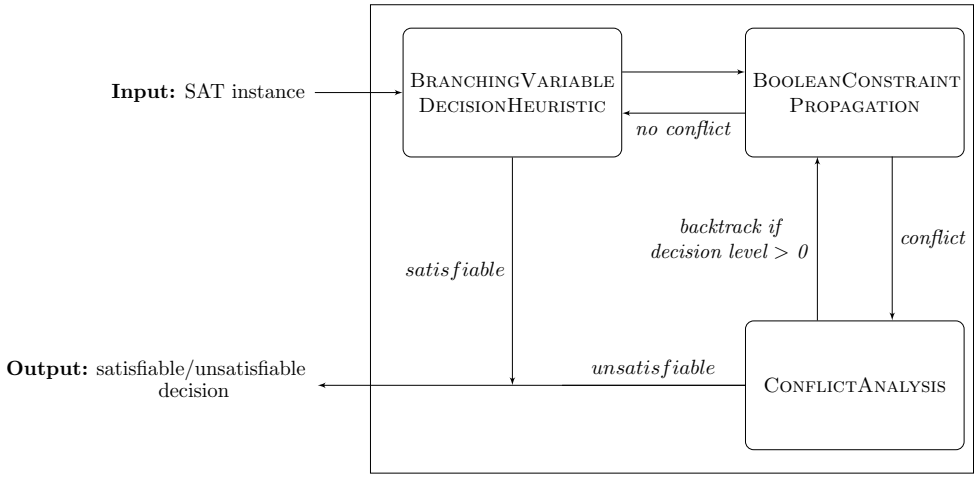


Figure 1. A general framework for CDCL-based SAT solvers.

3.1. CDCL Solvers in SAT Competitions.

The state of the art in CDCL SAT solvers can be thought of as the solvers that have participated in recent SAT competitions [5]. Most participating CDCL SAT solvers typically include at least one version or hack (To hack a base solver means to improve the solver by making only minor changes to its source code.) of MiniSat [32], Glucose [8], CryptoMiniSat [33], CaDiCaL [34], CoMiniSatPS [35], or more often, the most recent SAT competition(s) winners (e.g., [12,34]). For the latter to show an improvement of a CDCL SAT solver with a new heuristic, the solver must be compared to the base solver without any modifications. Table 1 shows the configuration of the winning CDCL SAT solvers used in this study (MapleLCMDistChronoBT-DL-v3 and LSTech are selected as the base solvers. In addition to the base solvers, the proposed solvers were evaluated against Relaxed_LCMDCBDL_newTech, Kissat_GB, and Kissat_MAB).

MapleLCMDistChronoBT-DL-v3 [12] is based on the winner of the 2018 SAT competition, MapleLCMDistChronoBT [36], and augmented with duplicate learnt heuristic. In

particular, Kochemazov et al. [12] improved the three-tier clause management by persisting additional clauses through the hash-based detection of repeatedly learnt clauses. They presented their solver MapleLCMDistChronoBT-DL-v3 in the 2019 SAT Race.

Relaxed_LCMDCBDL_newTech, a relaxed variant of MapleLCMDistCBT-DL, was first introduced in the 2020 SAT competition [15]. Basically, the idea is to relax the backtracking by protecting promising partial assignments from being pruned. Specifically, during the search, whenever a node corresponding to a promising assignment is reached, the algorithm enters a non-backtracking stage (under some conditions); this leads to a complete assignment, which is fed to an SLS solver to search for a solution nearby. In the 2021 SAT competition, its variant LSTech [37] showed a good performance especially on satisfiable instances [37].

The CDCL solver Kissat [34] is another popular base SAT solver that placed first in the 2020 SAT competition main track. Kissat is a low-level re-implementation of CaDiCaL that features improved data structures, better scheduling of inprocessing, and optimized algorithms and implementation. One configuration of Kissat is Kissat_SAT that targets satisfiable instances. Kissat_SAT is the base solver of Kissat_GB [38] and Kissat is the base of Kissat_MAB [39]. Kissat_GB bumps the variables based on glue centrality of glue variables. Kissat_MAB was augmented with the CHB decision heuristic as specified in [13]. The solver incorporates a reinforcement learning technique under the Multi Armed Bandit (MAB) framework that combines the VSIDS and the CHB branching heuristics by adaptively choosing the relevant heuristic at each restart using the Upper Confidence Bound (UCB) strategy.

3.2. Algorithms for Computing Backbones and Backdoors

Throughout the literature, many algorithms for backbone computation have been proposed. To begin with, Kaiser and Küchlin [40] proposed three backbone computation algorithms based on model enumeration and SAT testing. To compute backbones, Climer and Zhang [41] proposed a graph reduction technique called limit crossing. Janota et al. [26] computed backbones based on enumerating implicants and iterative SAT testing and optimizations with calls to a CDCL SAT solver. The work of Previti and Järvisalo [42] follows the idea of enumerating implicants in [26], but it differs in terms of using a CDCL SAT solver with preferences. Zhang et al. [43] suggested three sets of filtering optimization heuristics to improve the performance of the iterative SAT testing method for backbone computation. Dequen and Dubois [24] introduced a heuristic for backbone computation. The heuristic was incorporated into the DPLL solver, *k*cnfs, to encourage branching on backbone variables. Experimental results indicated that *k*cnfs performed well on random *k*-CNF formulas [5]. Wu [44] applied machine learning to optimize the values of the branching variables in MiniSat [32]. Experimental results confirmed that the solver managed to set on average 78% of the backbones correctly. The solver did reduce conflicts; however, the long preprocessing time outweighed the decrease in runtime.

Many algorithms for backdoor computation have been proposed in the literature. Williams et al. [18] computed strong backdoors in several industrial instances. Experimental results indicated that the size of the backdoor is close to zero. The authors also investigated the relatedness between backdoors, restarts, and the heavy-tailed distribution phenomena. They suggested that backdoors with sizes near-zero lead to runtime distributions that are lower bounded by heavy tails. This led Williams et al. [18] to hypothesize that SAT solvers that are effective in solving industrial SAT instances exploit backdoors. Kilby et al. [20] concluded that strong backdoors seem to be correlated with problem hardness on random 3-CNF formulas, whereas this was not observed for weak backdoors. Gregory et al. [21] studied weak backdoors for random and crafted instances. They observed that backdoor values for crafted instances are close to zero. Moreover, the authors discovered that when clause learning is enabled, the average backdoor size decreases. This is of interest because modern CDCL solvers implicitly exploit the backdoor structure. Zulkoski et al. [19] computed weak backdoors for all SAT categories on benchmark instances from 2009–2014 SAT competitions.

Experiments concluded that weak backdoors are hard to compute and small for all instance categories. Dilkina et al. [45] extended traditional backdoors to LS backdoors to take advantage of clause learning [29] during the search performed by a CDCL SAT solver. Experiments showed that LS backdoors are exponentially smaller than traditional strong backdoors on mixed integer programming SAT instances [46]. Overall, experiments on instances from all SAT categories have reported a near-zero backdoor size. Zulkoski et al. [19] extended the notion of LS backdoors to allow restarts by introducing the concept of LSR backdoors. The results confirmed that industrial instances indeed appear to have smaller LSR backdoor sizes compared to random instances. Zulkoski et al. [19] observed that the number of LSR backdoors that have been computed are of twice the instances of weak backdoors. Zulkoski et al. [19] concluded that LSR backdoor sizes are larger compared to weak backdoors. However, weak backdoors are harder to compute.

Table 1. The configuration of CDCL SAT solvers considered in this study. In the last column, under SAT competition rank, the SAT, UNSAT, or ALL indicate the type of track, which is either satisfiable, or unsatisfiable, or both, respectively.

CDCL Solver	Base Solver	Decision Heuristic	Restart Heuristic	Backtracks	Improvements	SAT Competition Rank
MapleLCM DistChronoBT -DL-v3 [12]	MapleLCM DistChronoBT	LRB and VSIDS	• Luby restarts for LRB • Glucose-style restarts for VSIDS	ChronoBT	• Duplicate learnts heuristic • Deterministic LRB-VSIDS switching	• Main track ALL/UNSAT 1st (2019) • Main track SAT 2nd (2019)
Relaxed LCMDCBDL-newTech [15]	MapleLCMDist-ChronoBT-DL	LRB and VSIDS	• Luby restarts for LRB • Glucose-style restarts for VSIDS	ChronoBT	• Relaxed CDCL approach • Probability Based Phase Saving	• Main track ALL 2nd (2020) • Main track SAT 1st (2020)
Kissat_MAB [39]	Kissat	VSIDS and CHB	• Infrequent restarts for SAT mode • Frequent restarts for UNSAT mode	ChronoBT	Multi-Armed Bandit framework which combines VSIDS and CHB	Main track ALL/SAT 1st (2021)
Kissat_gb [38]	Kissat_SAT	VSIDS and CHB	• Infrequent restarts for SAT mode • Frequent restarts for UNSAT mode	ChronoBT	Glue Bumping (GB) based on glue centrality of glue variables	Main track ALL/SAT 3rd (2021)
LSTech [37]	Relaxed LCMDCBDL-newTech	LRB and VSIDS	• Luby restarts for LRB • Glucose-style restarts for VSIDS	ChronoBT	Relaxed CDCL approach based on number of restarts	Main track SAT 2nd (2021)

3.3. *MapleLCMDistChronoBT-DL-v3 and LSTech*

The base solvers considered in this study, MapleLCMDistChronoBT-DL-v3 and LSTech, are described in the following section (The reader is referred to Appendix B.1 for a justification of the selection of base solvers.). Both solvers follow the general CDCL framework (A general framework for a CDCL solver is provided in Appendix C.1.). We specifically discuss the branching decision heuristic augmented in both solvers, as the proposed improvements are focused on it (A justification for choosing the branching decision heuristic to exploit backbones in order to improve the base solvers is described in Appendix B.2. In addition, Appendix C.2 provides the algorithm for the decision heuristic VSIDS implemented in Maple-based series SAT solvers.).

Algorithm 1 describes a top-level implementation for MapleLCMDistChronoBT-DL-v3. There are three decision heuristics augmented in it: Dist, VSIDS, and LRB. For the first 50,000 conflicts, the solver branches using the Dist heuristic (lines 18–19). After the

first 50,000 conflicts, the solver branches based on a deterministic LRB/VSIDS switching strategy (lines 21–25). It starts from LRB and switches each time the number of propagations since the last switch exceeds a threshold value. This value is initially set to 30,000,000 propagations; then, it increases by 10% with every switch (line 27).

A top-level implementation for LSTech SAT solver is presented in Algorithm 2. As the Dist heuristic did not improve the performance of the solver, the developers removed it. The solver switches based on a new deterministic restart-based strategy (lines 35–41). Each time the solver loops a threshold value, it will switch between VSIDS and LRB once. This threshold value is set to 500 (line 35). In addition, LSTech relaxes the backtracking process for protecting promising partial assignments (lines 33–34). When the solver reaches a promising partial assignment, it enters a non-backtracking stage until it obtains a full assignment (line 34). The SLS solver, CCAr, is then called every number of restarts (*sthreshold*) in order to find a satisfying assignment close to the full assignment (lines 16–28). This value of *sthreshold* is set at 300, but if the SLS solution does not improve, it increases; otherwise, it decreases, keeping the value above 300 (lines 24–27). If the SLS solver fails to find a satisfying assignment within certain limits, then the solver goes back to where it was interrupted (line 28).

Algorithm 1: Pseudocode for top-level implementation of MapleLCMDistChronoBT-DL-v3.

```
   Input: φ is a CNF formula with variables x ∈ X
   Output: satisfiable and solution, otherwise unsatisfiable
1  INITIALIZE()
2  while notALLVARIABLESASSIGNED() do
3      (φ, status, #propagations)←BOOLEANCONSTRAINTPROPAGATION(φ,solution)  // returns the status true
         if there is a conflict, false otherwise, and the number of propagations
4      if status is conflict then
5          #conflicts ← #conflicts + 1
6          (blevel, learntClauseVars, conflictSideVars) ←CONFLICTANALYSIS()  // build the learnt clause φ
             ← φ ∨ learntClause and return the new decision level
7          AFTERCONFLICTANALYSIS(learntClauseVars, conflictSideVars)
8          if blevel is equal to 0 then
9              return unsatisfiable
10         else
11             solution ←BACKTRACK(blevel,solution)              // backtrack to the new decision level
12             decisionLevel ← blevel
13             ONUNASSIGN(x)   // called when variable x is unassigned by backtracking or restart
14     else if restart condition is triggered then
15         solution←BACKTRACK(0,solution)                        // backtrack to the new decision level
16         ONUNASSIGN(x)        // called when variable x is unassigned by backtracking or restart
17     else
18         if #conflict < 50,000 then
19             branchMode ← Dist
20         else
21             if #propagations ≤ threshold                  // threshold value initialized to 30,000,000
22              then
23                 branchMode ← LRB
24             else
25                 branchMode ← VSIDS
26             if branchMode has switched to a new heuristic then
27                 threshold ← threshold + threshold × 0.1
28                 #propagations ← 0      // #propagations is reset to zero each time branchMode has
                     switched to a new heuristic
29         x ←PICKBRANCHINGVARIABLE()            // the decision heuristic is based on branchMode
30         ONASSIGN(x)           // called when variable x is assigned by branching or propagations
31         decisionLevel ← decisionLevel + 1              // increment decision level due to new decision
32         solution ← solution ∪ (x,value)
33 return (satisfiable,solution)
```

Algorithm 2: Pseudocode for top-level implementation of LSTech.

```
Input: φ is a CNF formula with variables x ∈ X
Output: satisfiable and solution, otherwise unsatisfiable
INITIALIZE()
while not ALLVARIABLESASSIGNED() do
    (φ,status,#propagations)←BOOLEANCONSTRAINTPROPAGATION(φ,solution)   // returns the status true if there is a conflict, false otherwise, and the number of propagations
    if status is conflict then
        #conflicts ← #conflicts + 1
        (blevel, learntClauseVars, conflictSideVars)←CONFLICTANALYSIS()   // build the learnt clauses φ ← φ ∨ learntClause and return the new decision level
        AFTERCONFLICTANALYSIS(learntClauseVars, conflictSideVars)
        if blevel is equal to 0 then
            return unsatisfiable
        else
            BACKTRACK(blevel,solution)                          // backtrack to the new decision level
            decisionLevel← blevel
            ONUNASSIGN(x)    // called when variable x is unassigned by backtracking or restart
    else if restart condition is triggered then
        #restarts ← #restarts + 1
        if #restarts − sLastrestart is equal to sthreshold // sLastrestart is initialized 0 and sthreshold to 300 then
            (result, slsBestSolution)←CCAr(φ,promissingFullAsgmt) // SLS solver is called to find the best assignment near promissingFullAsgmt
            if result equals true    // SLS solver found a satisfying assignment, so solution is found, otherwise resume normal solver mode
            then
                return (satisfiable,slsBestSolution)
            else
                sLastrestart ← #restarts
                if slsBestSolution is an improved assignment then
                    sthreshold ← sthreshold − 300                          // keep sthreshold > 300
                else
                    sthreshold ← sthreshold + 300
                resume normal mode
        BACKTRACK(0,solution)                                  // backtrack to the new decision level
        ONUNASSIGN(x)          // called when variable x is unassigned by backtracking or restart
    else
        if nonBacktrack condition is triggered                                  // enter a non-bactracking stage
        then
            promissingFullAsgmt ← NONBACKTRACK()  // build a promising full assignment within a time limits, otherwise resume normal solver mode
        if #restarts − bLastrestart > bthreshold      // bLastrestart is initialized to zero and bthreshold to 500.
        then
            if branchMode equals VSIDS then
                branchMode ← LRB
            else
                branchMode ← VSIDS
            bLastrestart ← #restarts                                    // bLastrestart is reset to #restarts
        x←PICKBRANCHINGVARIABLE()              // the decision heuristic is based on branchMode
        ONASSIGN(x)              // called when variable x is assigned by branching or propagations
        decisionLevel ← decisionLevel + 1               // increment decision level due to new decision
        solution ← solution ∪ (x, value)
return (satisfiable,solution)
```

4. The Proposed CDCL SAT Solvers

A description of the proposed CDCL SAT solvers that exploit backbones and backdoors is presented in this section. To begin with, for the purpose of guiding the CDCL SAT solver to branch on backcbones/backdoors, two low-overhead heuristics are proposed for computing backbones and backdoors. They are described in the following subsection.

4.1. Backbone and Backdoor Computational Heuristics

This section details the proposed heuristics for computing the backbone and backdoor sets: backbone low-overhead (BBLO) heuristic and backdoor low-overhead (BDLO) heuristic. Both heuristics are designed based on relevant features of the backbone and backdoor while having low computational overhead. As described below, this trade-off between

accuracy and computational time is motivated by the work of [47]. It should be noted that these heuristics are not intended to be standalone heuristics for computing the backbones and backdoors. Rather, they are incorporated within the CDCL SAT solver, in particular the branching heuristic, to encourage the solver to branch on these variables.

4.1.1. The BBLO Heuristic

The BBLO heuristic was inspired by the low-overhead computation of backbones in [47]. Menai and Batouche [47] proposed a backbone-based co-evolutionary algorithm for the partial maximum satisfiability problem guided by the estimated backbone literals of the problem. That is, literals that are set to the same value on each run will be considered backbones; otherwise, they will not. BBLO adopts a similar concept except that local search runs are the search tree trails and non-backbone literals are identified in each run (A trail is the partial assignment that represents the current path in the search tree produced by a CDCL SAT solver.). This process is repeated for each restart.

The BBLO heuristic is presented in Algorithm 3. It is called during every restart procedure in Algorithms 1 and 2. At the first call of the Algorithm 3, all variables in *list* are marked as undefined. Upon completing a trail, the initial value of the *list* is its final value of the previous trail. For each assigned variable (line 3), the heuristic determines if it is not a backbone literal whenever a variable has a different assignment than the previous trail (lines 7 and 8). At the end, all assigned variables in *List* are moved as literals to the backbone set.

Algorithm 3: A backbone-based low-overhead computational heuristic (BBLO).

Input: ϕ: CNF formula with a set of variables X
Output: $BackBones_\phi$: a set of backbone literals for ϕ
Data: $List_\phi$: a list of size $|X|$ where each entry takes one of the values: *true*, $false$, $undefined$, or *notBackBone*. $List_\phi$ is initialized to $undefined$ in the first call of Algorithm 3 and its final value is its initial value upon completing a trail.

1 **if** *trail is completed;* `// a trail completion triggers the computation of the backbone set`
2 **then**
3 **for** $x \in Assigned_\phi$; `//` $Assigned_\phi$ `is the set of assigned variables so far in` ϕ.
4 **do**
5 **if** $List_\phi[x]$ *is* $undefined$ **then**
6 $List_\phi[x] \leftarrow x_{val}$; `//` x_{val} `is the assignment value of variable` x; `either` *true* `or` $false$
7 **else if** $List_\phi[x]$ *is not marked notBackBone and* $List_\phi[x]$ *is not equal to* x_{val} **then**
8 $List_\phi[x] \leftarrow notBackBone$
9 Copy literals in $List_\phi$ to $BackBones_\phi$
10 **return** $BackBones_\phi$

4.1.2. The BDLO Heuristic

The BDLO heuristic is inspired by the findings in [20,21] as well as our findings in Appendix A that show experimentally the near-zero overlap between backbone and LSR backdoor variables on all benchmark categories. Consequently, the BDLO heuristic computes the backdoor variables from the backbones in Algorithm 3. As computed in Algorithm 4, backdoor variables are those that are not backbones (line 9).

Algorithm 4: A backdoor-based low-overhead computational heuristic (BDLO).

Input: ϕ: CNF formula with a set of variables X
Output: $BackDoors_\phi$: a set of backdoor variables for ϕ
Data: $ListBB_\phi$: a list of size $|X|$ that takes values *true*, *false*, *undefined*, and *notBackBone*, $ListBD_\phi$: a list of size $|X|$ that takes values *true*, *false*, and, *notBackDoor*. $List_\phi$ is initialized to *undefined* and $ListBD_\phi$ to *notBackDoor* in the first call of Algorithm 4 and their final value is their initial values upon completing a trail.

1 **if** *trail is completed;* `// a trail completion triggers the computation of the backbone set`
2 **then**
3 **for** $x \in Assigned_\phi$; `//` $Assigned_\phi$ `is the set of assigned variables so far in` ϕ
4 **do**
5 **if** $ListBB_\phi[x]$ *is undefined* **then**
6 $ListBB_\phi[x] \leftarrow x_{val}$; `//` x_{val} `is the assignment value of variable` x`; either` *true* `or` *false*
7 **else if** $ListBB_\phi[x]$ *is not marked notBackBone and* $ListBB_\phi[x]$ *is not equal to* x_{val} **then**
8 $ListBB_\phi[x] \leftarrow notBackBone$
9 $ListBD_\phi[x] \leftarrow x_{val}$
10 Copy assigned variables in $ListBD_\phi$ to $BackDoors\phi$
11 **return** $BackDoors\phi$

4.2. VSIDS Variants

We chose to exploit backbone and backdoor variables by extending the VSIDS heuristic (A detailed description of the VSIDS heuristic is provided in Appendix C.2, along with a justification for choosing the branching heuristic to improve the solver is given in Appendix B.2). The following is a summary of VSIDS's policy implemented in MapleLCMDistChronoBT-DL-v3 and LSTech:

1. Initialization: Each variable has a floating point number, called activity, which is initialized to 0.
2. Additive bump: Following a conflict analysis phase, the activities of all variables that led to a conflict are additively bumped (increased), typically by 1, if their decision levels are greater than the backtrack level; otherwise, they are bumped by 0.5.
3. Decision: The (unassigned) variable with the highest activity is chosen at each decision.
4. Multiplicative decay: All variables are periodically decremented by multiplying their activities by a constant $0 < Decay < 1$ called the multiplicative decay factor.

To exploit backbone and backdoor variables, the VSIDS heuristic was extended, specifically, the additive bump policy. We considered the following cases: when to bump, which variables to bump, and with what value to bump. Accordingly, six variants of the VSIDS heuristic were derived: VSIDS_BBsize_restart, VSIDS_BBsize_BBfreq_restart, VSIDS_BBsize_BBfreq_backtrack, VSIDS_BBsize_BBfreq_conflict, VSIDS_BBsize_BBfreq_restart_conflict, and VSIDS_BDsize_restart. Algorithms 5–10 are variants of the VSIDS, and throughout, we highlight only the added policies to the VSIDS implemented in Maple-based series SAT solvers (see Appendix C.2). For all algorithms, all variables' activities are initialized to zero. In addition, the backbones and backdoors are computed on every restart by calling BBLO or BDLO. The policy of each variant is detailed below:

1. VSIDS_BBsize_restart (Algorithm 5): Following a restart, the activities of the backbone literals computed in Algorithm 3 are additively bumped, typically by 1 (lines 4–5).

Algorithm 5: VSIDS_BBsize_restart a variant of the VSIDS decision heuristic.

```
   // Called when the solver restarts.
1  Procedure RESTART()
2      ...
3      BB_φ ←BBLO(φ)                          // compute backbone literals, Algorithm 3.
4      for x ∈ BB_φ do
5          bumpActivity(x, 1)    // bump the activities of every backbone by 1 after every
             restart.
6      ...
```

2. VSIDS_BBsize_BBfreq_restart (Algorithm 6): Following a restart, the activities of the backbone literals computed in Algorithm 3 are additively bumped typically by their frequencies (lines 4–6).

Algorithm 6: VSIDS_BBsize_BBFreq_restart a variant of the VSIDS decision heuristic.

```
   // Called when the solver restarts.
1  Procedure RESTART()
2      ...
3      BB_φ ←BBLO(φ)                          // compute backbone literals, Algorithm 3.
4      for x ∈ BB_φ do
           BBFreq_x ← (Σ_{i=1}^{|C|} 1_{x∈c_i,c_i∈C}) / |C|   // BBFreq_x is the ratio of the total number of occurrences
5            of x in φ to the total number of clauses.
           bumpActivity(x, BBFreq_x)        // bump the activities of all backbones by their
6            BBFreq_x after every restart.
7      ...
```

3. VSIDS_BBsize_BBfreq_backtrack (Algorithm 7): Following a backtrack, the activities of the backbone literals computed in Algorithm 3 are additively bumped typically by their frequencies (lines 3–4).

Algorithm 7: VSIDS_BBsize_BBFreq_backtrack a variant of the VSIDS decision heuristic.

```
   // Called when the solver backtracks.
1  Procedure BACKTRACK()
2      ...
3      for x ∈ BB_φ do
4          bumpActivity(x, BBFreq_x)        // bump the activities of all backbones by their
             BBFreq_x after every backtrack.
5      ...
   // Called when the solver restarts.
6  Procedure RESTART()
7      ...
8      BB_φ ←BBLO(φ)                          // compute backbone literals, Algorithm 3.
9      for x ∈ BB_φ do
           BBFreq_x ← (Σ_{i=1}^{|C|} 1_{x∈c_i,c_i∈C}) / |C|   // BBFreq_x is the ratio of the total number of occurrences
10           of x in φ to the total number of clauses.
11     ...
```

4. VSIDS_BBsize_BBfreq_conflict (Algorithm 8): Following a conflict, the activities of the backbone literals computed in Algorithm 3 that led to the learnt clause (including those in the learnt clause) are additively bumped typically by their frequencies (lines 3–6).

Algorithm 8: VSIDS_BBsize_BBfreq_conflict a variant of the VSIDS decision heuristic.

```
   // Called after a learnt clause is generated by the ConflictAnalysis() procedure.
1  Procedure AfterConflictAnalysis(conflictSideVars ⊆ X, learntClauseVars ⊆ X)
2      ...
3      for x ∈ (conflictSideVars ∪ learntClauseVars) do
4          if x ∈ BB_φ then
5              bumpActivity(x, BBFreq_x)        // bump the activities of backbones by their
                 BBFreq_x after every conflict.
6          else
7              ...                                          // Same as VSIDS in Appendix C.2
   // Called when the solver restarts.
8  Procedure Restart()
9      ...
10     BB_φ ←BBLO(φ)                                  // compute backbone literals, Algorithm 3.
11     for x ∈ BB_φ do
                                  Σ_{i=1}^{|C|} 1
12         BBFreq_x ← ------------------------------  // BBFreq_x is the ratio of the total number of occurrences
                       x∈c_i,c_i∈C / |C|
                   of x in φ to the total number of clauses.
13     ...
```

5. VSIDS_BBsize_BBfreq_restart_conflict (Algorithm 9): In combination of variant 2 and 4.

Algorithm 9: VSIDS_BBsize_BBfreq_restart_conflict a variant of the VSIDS decision heuristic.

```
   // Called after a learnt clause is generated by the ConflictAnalysis() procedure.
1  Procedure AfterConflictAnalysis(conflictSideVars ⊆ X, learntClauseVars ⊆ X)
2      ...
3      for x ∈ (conflictSideVars ∪ learntClauseVars)
4       do
5          if x ∈ BB_φ then
6              bumpActivity(x, BBFreq_x)        // bump the activities of backbones by their
                 BBFreq_x after every conflict.
7          else
8              ...                                          // Same as VSIDS in Appendix C.2
9      ...
   // Called when the solver restarts.
10 Procedure Restart()
11     ...
12     BB_φ ←BBLO(φ)                                  // compute backbone literals, Algorithm 3.
13     for x ∈ BB_φ do
                                  Σ_{i=1}^{|C|} 1
14         BBFreq_x ← ------------------------------  // BBFreq_x is the ratio of the total number of occurrences
                       x∈c_i,c_i∈C / |C|
                   of x in φ to the total number of clauses.
15         bumpActivity(x, BBFreq_x)        // bump the activities of all backbones by their
             BBFreq_x after every restart.
16     ...
```

6. VSIDS_BDsize_restart (Algorithm 10): Following a restart, the activities of the backdoor variables computed in Algorithm 4 are additively bumped typically by 1 (lines 4–5).

Algorithm 10: VSIDS_BDsize_restart a variant of the VSIDS decision heuristic.

```
// Called when the solver restarts.
Procedure RESTART()
    ...
    BD_φ ←BDLO(φ)                          // compute backdoor variables, Algorithm 4.
    for x ∈ BD_φ do
        bumpActivity(x, 1)  // bump the activities of every backdoor variable by 1 after
          every restart.
    ...
```

4.3. The CDCL SAT Solvers

To demonstrate that the six proposed VSIDS decision heuristics (Algorithms 5–10) contribute to the state of the art, we implemented each of them on top of two base CDCL SAT solvers, MapleLCMDistChronoBT-DL-v3 and LSTech. In doing so, three different versions of the six extended CDCL SAT solvers were developed. In version 1, during restarts/conflicts/backtracks, bumping occurs only in the VSIDS phase. In version 2, bumping during conflicts is only for the VSIDS phase. But, for restarts/backtracks the bumping occurs in both VSIDS and LRB phases (As the bump in VSIDS_BBsize_BBfreq_conflict occurs only during conflicts, it is the same for versions 1 and 2.). Version 3 is the same as version 1, except that in computing the bump with the backbone frequency, the polarity of variables is taken into account (Version 3 is not applicable to the variants VSIDS_BBsize_restart and VSIDS_BDsize_restart.). For convenience, we name each solver by starting with the base solver name, which is followed by the VSIDS variant, and then the version number, e.g., MapleLCMDistChronoBT-DL-v3_VSIDS_BDsize_restart_version1.

5. Performance Evaluation

5.1. Benchmarks and Experimental Setup

Experimental evaluation was performed on industrial benchmark instances drawn from 2002–2021 SAT benchmark instances. On the basis of Table 2, we can see that there are 32 industrial families out of which 12 are satisfiable, 5 are not, and 15 are both. For the purpose of reducing bias, each family contains the same number of satisfiable or/and unsatisfiable instances, seven, and almost the same values of backbone-related measures: backbone size, frequency, and coverage.

Table 2. A summary of industrial benchmarks used in the performance evaluation drawn from 2002–2021 SAT benchmark instances.

#families (satisfiable-unsatisfiable-both)	32 (12-5-15)
#instances	329
#satisfiable instances (with backbones-without backbones)	189 (185-4)
#unsatisfiable instances	140

The CDCL SAT solvers were evaluated on Shaheen II [48], which is a supercomputer at King Abdullah University of Science and Technology (KAUST). Each node of the cluster is equipped with 128 GB of DDR4 memory and a dual CPU based on 16-core Intel Haswell processors running at 2.3 GHz. In accordance with the literature, the time limit for solving each instance was set at 3600 s. A comparison is presented here between the extended proposed CDCL SAT solvers and their counterpart bases as well as three state-of-the-art winning solvers: RelaxedLCMDCBDL_newTech [15], Kissat_GB [38], and Kissat_MAB [39]. The primary metric is the number of solved instances along with the PAR-2 score. The PAR-2 score is the sum of runtimes for all solved instances and twice the timeout for each unsolved instance, so that a lower score is better.

5.2. Results

The performance results of the six proposed VSIDS extensions implemented in 15 variants of MapleLCMDistChronoBT-DL-v3 and LSTech are presented in Table 3. Each entry represents the number of solved instances. Bold values indicate that the variant performed better than the base solver. MapleLCMDistChronoBT-DL-v3 and LSTech solved 302 and 308, respectively, out of 329 industrial instances. The best performing variant for each base solver is MapleLCMDistChronoBT-DL-v3_VSIDS_BBsize_BBfreq_restart_version2 (Maple_VBBsfr_v2) and LSTech_VSIDS_BBsize_BBfreq_restart_conflicts_version1 (LSTech _VBBsfrc_v1). Maple_VBBsfr_v2 solved 307 instances, which was five more than the base solver. With LSTech_VBBsfrc_v1, four more instances were solved than with the base solver. For the solvers based on VSIDS_BDsize_restart, the only variant that improved is LSTech_VSIDS_ BDsize_restart_version1. Furthermore, note that the solver variants based on backtracks (row number 3) did not perform well.

Table 3. Performance results of the CDCL SAT variants on industrial benchmark instances. The versions 1, 2, and 3 are explained in Section 4.3. The base solver Maple is short for MapleLCMDistChronoBT-DL-v3.

	VBS [1]	**Number of Solved Industrial Instances** **329**					
		Version 1		**Version 2**		**Version 3**	
	Base Solver	**Maple** **302**	**LSTech** **308**	**Maple** **302**	**LSTech** **308**	**Maple** **302**	**LSTech** **308**
1	VSIDS_BBsize_restart	301	**309**	301	**310**	— [3]	
2	VSIDS_BBsize_BBfreq_restart	302	304	**307**	308	**305**	**309**
3	VSIDS_BBsize_BBfreq_backtrack	296	305	290	302	299	302
4	VSIDS_BBsize_BBfreq_conflict	**303**	308	— [2]		299	308
5	VSIDS_BBsize_BBfreq_restart_conflict	300	**312**	301	**310**	302	**311**
6	VSIDS_BDsize_restart	299	**310**	302	308	— [3]	

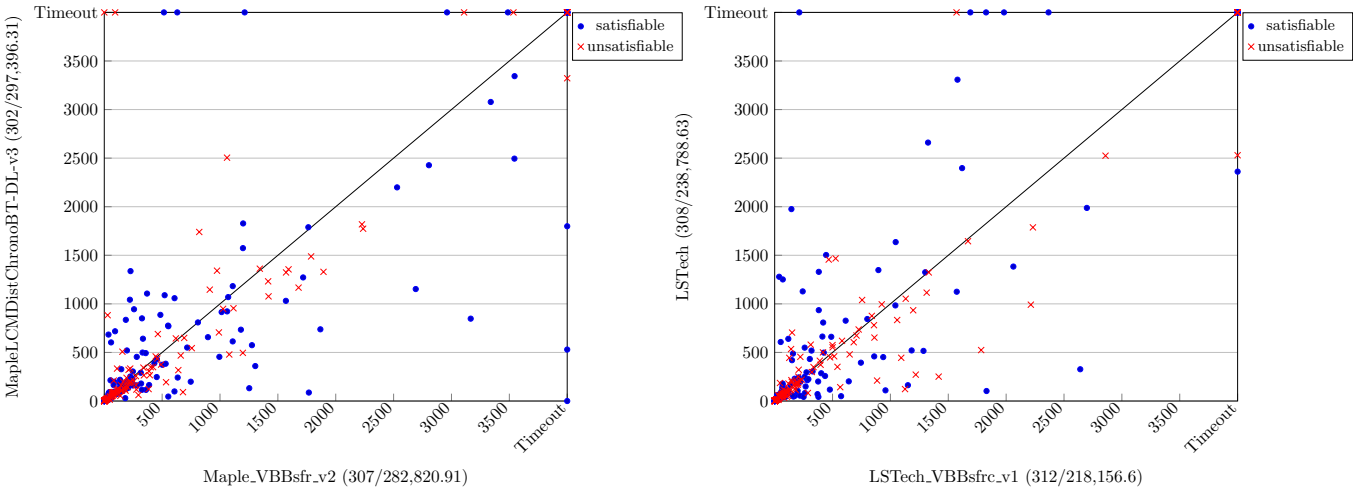
[1] Virtual Best Solver; [2] Implementation of version 1 and 2 for this variant is the same, since this variant does not bump variables during restarts; [3] Version 3 is not applicable, since this variant does not bump variables based on the backbone frequencies.

The scatter plots shown in Figure 2a,b detail performance comparisons of Maple_ VBBsfr_v2 and LSTech_VBBsfrc_v1 to their base solvers. Each improved variant solved more instances and achieved a lower PAR-2 score than its base counterpart.

Table 4 and Figure 3 report runtime results of Maple_VBBsfr_v2 and LSTech_VBBsfrc_v1 compared to state-of-the-art CDCL SAT solvers on the whole benchmark instances in Table 2. It is observed that LSTech_VBBsfrc_v1 scored 312, which is one instance more than the number of instances solved by Kissat_MAB, the winner of the 2021 SAT competition. According to Table 4 and Figure 4a,b, Maple_VBBsfr_v2 showed improvements on both satisfiable and unsatisfiable benchmarks, while LSTech_VBBsfrc_v1 showed improvements on satisfiable benchmarks.

Table 4. Performance evaluation of CDCL SAT solvers on industrial instances from the 2002–2021 SAT competitions. #SAT (#UNSAT) denotes the number of solved satisfiable (unsatisfiable) instances. Appendix D reports the results in detail.

VBS	**#SAT** **189**	**#UNSAT** **140**	**Total** **329**
MapleLCMDistChronoBT-DL-v3	175	127	302
Maple_VBBsfr_v1	177	130	307
LSTech	179	129	308
LSTech_VBBsfrc_v2	**183**	129	**312**
Relaxed_LCMDCBDL_newTech	178	128	306
Kissat_GB	166	**131**	297
Kissat_MAB	**183**	128	311



(**a**) (**b**)

Figure 2. (**a**,**b**) Performance comparison of Maple_VBBsfr_v2 and LSTech_VBBsfrc_v1 to their base solvers. The results are on the whole 32 industrial benchmarks. For the x and y axes, the labels are of the form a(b/c), where a is the solver name, b is the total number of instances solved, and c is the PAR-2 score. In subfigure (**b**), for example, a point (1000, 1200) means that LSTech_VBBsfrc_v1 took 1000 s to solve the specified instance, whereas LSTech took 1200 s.

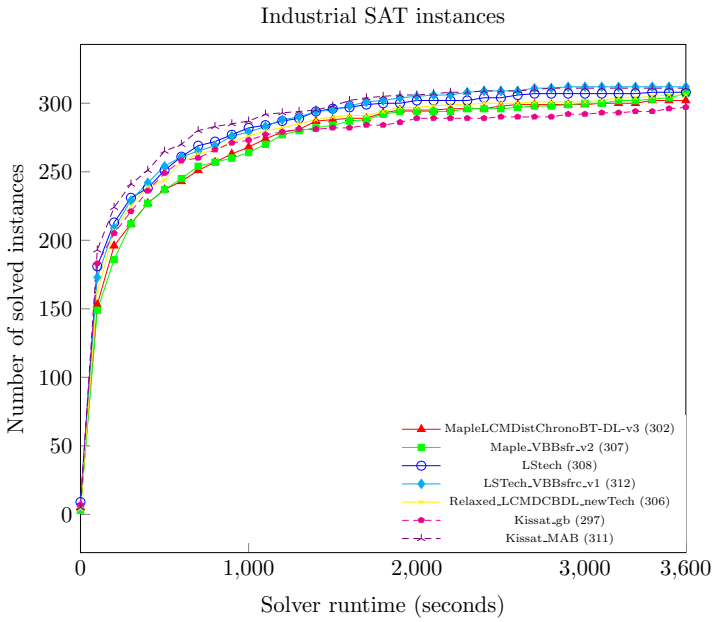


Figure 3. Performance results of Maple_VBBsfr_v2 and LSTech_VBBsfrc_v1 with state-of-the-art CDCL SAT solvers. Results are on the whole industrial benchmarks. A point (1500, 80) is interpreted as follows: there are 80 instances that took less than 1500 s to solve with the respective SAT solver.

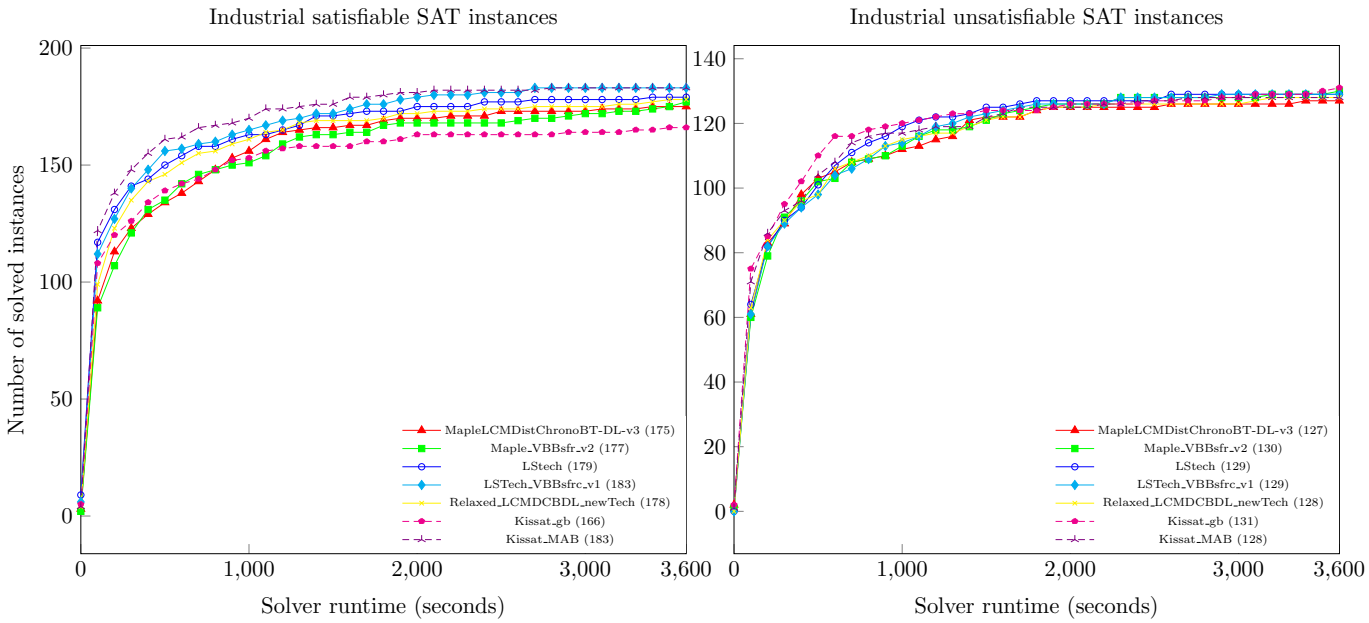


Figure 4. Performance results of Maple_VBBsfr_v2 and LSTech_VBBsfrc_v1 with state-of-the-art CDCL SAT solvers. Subfigures (**a**,**b**) illustrate results on satisfiable and unsatisfiable industrial benchmarks, respectively.

6. Discussion

Backbones and backdoors have been introduced for over two decades, but the current state of the art of CDCL SAT solvers [17,49] shows limited evidence of their use to improve CDCL SAT solvers. Among the few is Dequen and Dubois' work [24]. The backbone-guided decision heuristic they proposed was heavily dependent on unit propagation, which consequently added a significant overhead to the solver performance. It should be noted that early in our work, we have implemented the backbone guided decision heuristic [24] on top of MapleLCMDistChronoBT-DL-v3 to compute backbones; however, the solver performed poorly and never beat the base solver. In addition, it was demonstrated in [19,50] that it is difficult to build a solver that detects backdoors due to the difficulty of computing them. In this regard, we have proposed the BBLO and BDLO heuristics for computing backbones and backdoors. As far as backdoors, they are calculated under the assumption of no overlap between backbones and backdoors, which is only an experimental observation.

According to the performance evaluation in Section 5, both variants Maple_VBBsfr_v2 and LSTech_VBBsfrc_v1 have outperformed the base solvers on solving satisfiable benchmarks, that is, instances with backbones. Therefore, improvements in VSIDS have led to improved performance for both variants in solving more satisfiable instances. It is therefore evident that VSIDS is not only superior at solving unsatisfiable instances [51] but could even be improved to solve satisfiable instances as well.

Further observations include the improved performance of Maple_VBBsfr_v2 with both satisfiable and unsatisfiable benchmarks, while LSTech_VBBsfrc_v1 only improved with satisfiable benchmarks. Evidence suggests that the BBLO heuristic could assist the CCAr solver in finding an earlier solution by identifying promising partial assignments.

7. Conclusion and Future Work

In this study, we performed a set of modifications to the VSIDS decision heuristic to exploit backbones and backdoors. To guide the CDCL SAT solver to branch on backbone/backdoor variables, we proposed low-overhead heuristics to compute them. We implemented 15 variants on top of two popular state-of-the-art award-winning CDCL solvers, MapleLCMDistChronoBT-DL-v3 and LSTech. The variants were evaluated on

32 industrial families from 2002 to 2021 SAT competitions. Results showed that bumping the backbone and backdoor variables during the branching heuristic improved both solvers' performance. In particular, the variant of LSTech that augmented the VSIDS decision heuristic to exploit the backbone size and frequency and bumped during restarts and conflicts solved more industrial instances compared to the best state-of-the-art CDCL solvers. In the future, we would like to assess the accuracy of the proposed backbone and backdoor heuristics and to investigate the possibility of any improvements. A further research goal would be to modify other decision heuristics, such as LRB and CHB. Moreover, we would like to conduct a larger study to include broader bases of CDCL SAT solvers and benchmarks.

Author Contributions: Conceptualization, T.A.-Y. and M.E.B.A.M.; methodology, T.A.-Y. and M.E.B.A.M.; software, T.A.-Y.; validation, T.A.-Y. and M.E.B.A.M.; formal analysis, T.A.-Y. and M.E.B.A.M.; investigation, T.A.-Y. and M.E.B.A.M.; resources, T.A.-Y. and M.E.B.A.M.; data curation, T.A.-Y.; writing—original draft preparation, T.A.-Y.; writing—review and editing, T.A.-Y. and M.E.B.A.M.; supervision, M.E.B.A.M. and H.M. All authors have read and agreed to the published version of the manuscript.

Funding: The authors would like to thank the Deanship of Scientific Research (DSR) in King Saud University for funding and supporting this research through the initiative of DSR Graduate Students Research Support (GSR).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The datasets generated during the current study are not publicly available but are available from the corresponding author on reasonable request.

Acknowledgments: We are grateful to the KAUST Supercomputing Laboratory for the use of Shaheen II (https://www.hpc.kaust.edu.sa (accessed on 15 January 2020)). We thank them for providing the computational resources and professional support for carrying out this work.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. The Evidence of Boolean Structural Measures for SAT

Boolean structure measures well known in the literature are: phase transition, backbone size, backdoor size, and backbone/backdoor variable overlap size [25]. In view of the fact that the phase transition is only located for random and industrial-like instances, we focus on characterizing SAT instances based on backbones and backdoors. Interest in backbone and backdoor size was originally motivated to relate them to problem hardness and the performance of the SAT solver [20]. Backbones have been used to guide the variable decision heuristics for SAT solvers [24]. Insight into the heavy-tailed phenomena observed in backtrack heuristics comes from considering backdoor variables. Intuitively, backtrack heuristics appear to be lucky on certain runs due to small (near-zero) backdoor sizes. Consequently, backdoor variables are identified early and set correctly [31,49,52]. Boolean structural measures examined are the backbone size, backdoor size, and backbone/backdoor variable overlap size. We also introduced three new related measures: backbone frequency, backbone coverage, and backdoor coverage. All measures are defined in Section 2 except for backbone coverage and backdoor coverage, which are defined below:

Definition A1 (Backbone coverage)**.** *Backbone coverage is the percentage proportion of clauses that contains backbone literals in a CNF formula (after CNF formula simplification), denoted* $BBcvg_\phi$*. Backbone coverage is expressed as follows:*

$$BBcvg_\phi = \frac{\sum\limits_{\substack{i=1,\\ \exists x_b \in BB_\phi \Leftrightarrow x_b \in C_i \\ s.t.\ C_i \in C}}^{|C|} 1}{|C|} \times 100 \tag{A1}$$

Definition A2 (Backdoor coverage)**.** *Backdoor coverage is the percentage number of clauses that contain backdoor variables in a CNF formula (after CNF formula simplification), denoted $BDcvg_\phi$. This is formulated as follows:*

$$BDcvg_\phi = \frac{\sum\limits_{\substack{i=1,\\ \exists x \in BD_\phi \Leftrightarrow x \in C_i \\ s.t.\ C_i \in C}}^{|C|} 1}{|C|} \times 100 \tag{A2}$$

Appendix A.1. Backbone and Backdoor Computation

To demonstrate the presence of backbones and backdoors for SAT instances, we calculated these measures across the entire benchmark sets. Backbones were computed exactly by the tool in [42]. This tool is based on repeated calls to MiniSat. Based on the backbone sets, the frequencies and coverages of the backbone sets were calculated as shown in Equations (3) and (A1).

LSR-backdoors were computed using a tool developed by [19] called LaSeR. Due to the difficulty of exactly computing the LSR-backdoors, the tool was configured to compute an upper bound on the size of the minimal LSR-backdoor. The tool is built on top of the MapleSAT solver [14]. After the backdoor sets were computed, the backdoor coverages were calculated, as shown in Equation (A2). Finally, the backbone/backdoor overlap was computed based on Equations (6) and (7).

Appendix A.2. Benchmarks and Experimental Setup

Experiments were conducted on Shaheen II [48]. A timeout period of 72 h was set to compute all backbone-related measures (backbone size, backbone frequency, and backbone coverage) and similarly for backdoor-related measures (backdoor size, backdoor coverage, and backbone/backdoor variable overlap size) for each of the 7779 instances (see Table A1) (We were unable to locate benchmarks for 2004, 2005, or 2017).

Table A1. Number of instances included in experiments drawn from 2002–2020 SAT benchmarks. The symbol × means that no instances were submitted under the corresponding category.

Year	Industrial	Crafted	Random
2002	209	541	97
2003	69	259	138
2006	70	×	×
2007	161	16	406
2008	62	×	×
2009	165	159	570
2010	100	×	×
2011	150	150	613
2012	208	53	600
2013	150	201	430
2014	58	121	225
2015	176	×	×
2016	182	200	240
2018	400	×	×
2019	200	×	×
2020	400	×	×

Appendix A.3. Experimental Results

We computed the backbones for all satisfiable instances presented in Table A1. In Figure A1, we report the number of satisfiable instances in each benchmark category with

and without backbones. In contrast to random instances, most satisfiable industrial and crafted instances preserve the backbone structure. It is due to the fact that the backbone sizes for most random instances are controlled by being set to zero prior to generation.

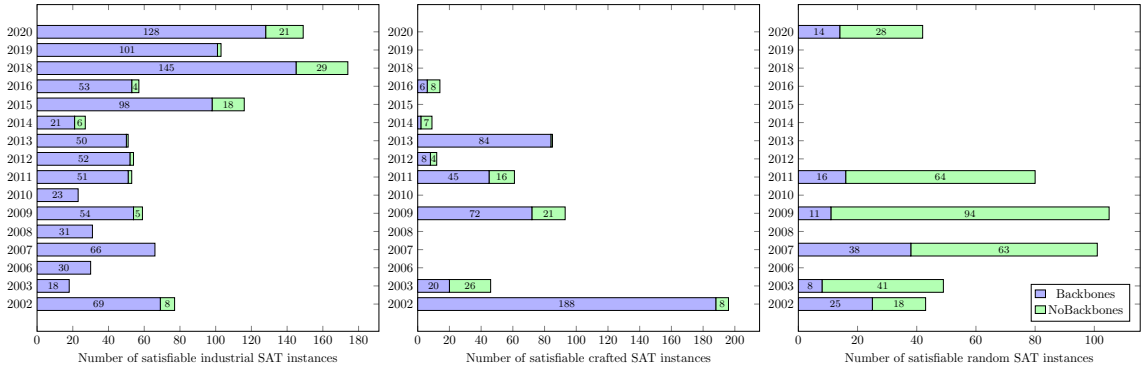


Figure A1. Number of instances with (w\o) backbones on 2002–2020 SAT benchmarks.

Structural measures related to backdoors are computed on all satisfiable and unsatisfiable SAT benchmark instances (see Table A1). Figure A2 displays the number of satisfiable and unsatisfiable instances with backdoors. One point to note is that the reported instances for backdoors are lower than those for backbones. This is due to the hardness of computing LSR backdoors [53]. The number of satisfiable instances with backbone/backdoor variable overlap is shown in Figure A3.

All Boolean measures are computed, along with their means and standard deviations. A summary of the results is presented in Table A2. Empirical results indicate that most random benchmarks have no backbones, whereas on average, industrial and crafted benchmark instances have small backbone sizes. The frequency of backbones is low for all benchmark categories. As for the backbone coverage, industrial and crafted benchmarks have higher coverages, on average, than random ones. In both crafted and random benchmark instances, the backdoor size and coverage are greater than those in the industrial category. Additionally, across all SAT benchmarks, there tends to be little overlap between backbone and backdoor variables.

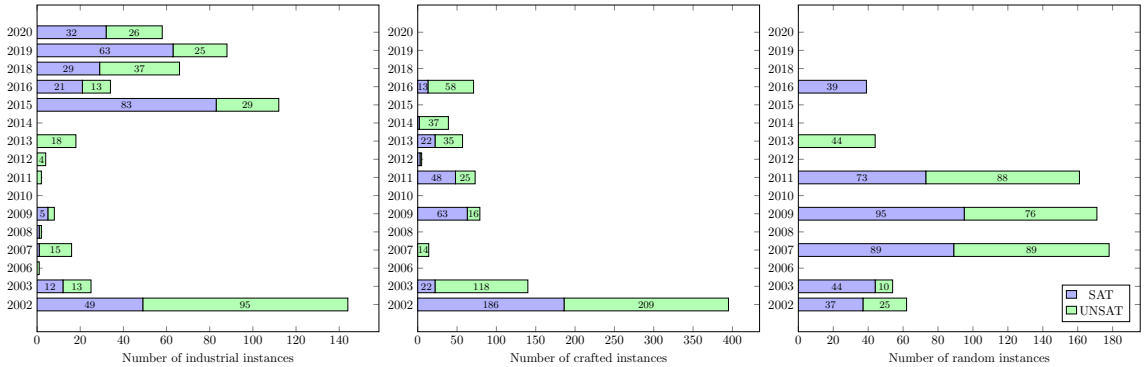


Figure A2. Number of satisfiable/unsatisfiable instances with backdoors on 2002–2020 SAT benchmark instances.

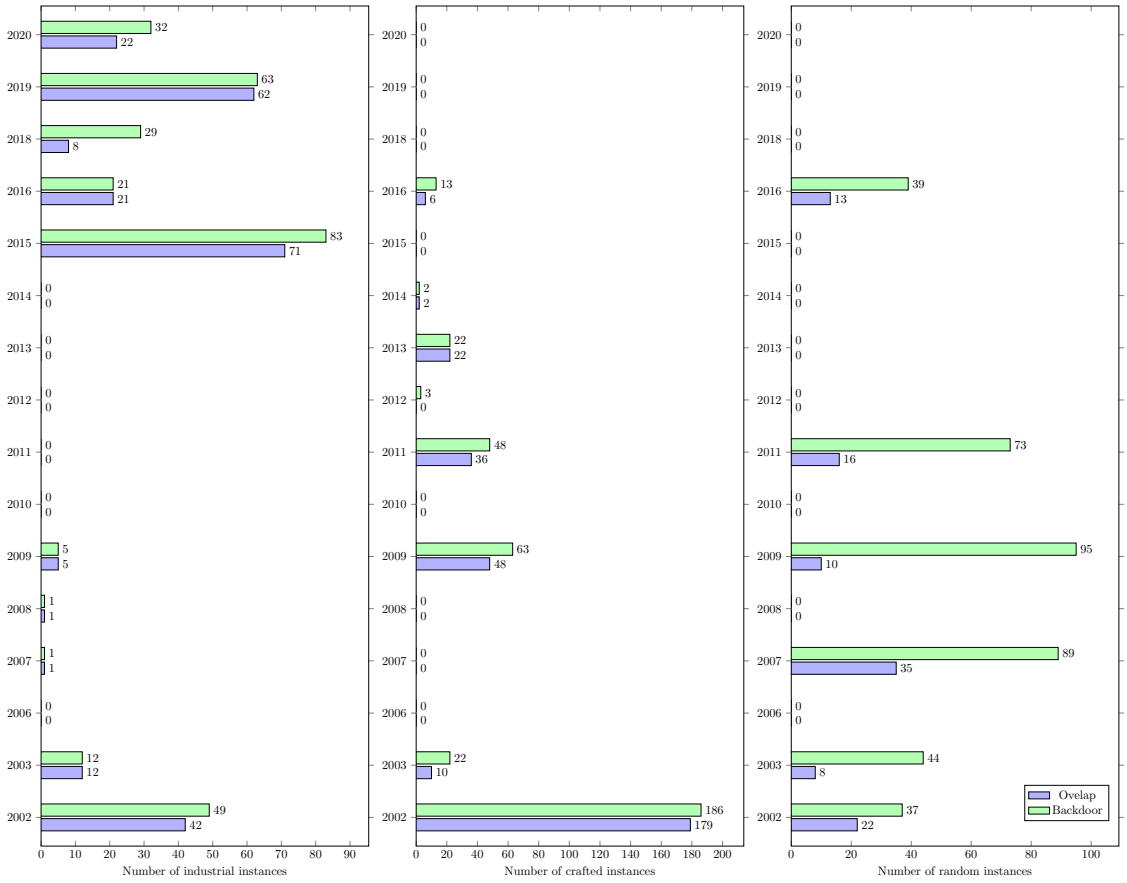


Figure A3. Number of instances with backbone/backdoor variable overlap on 2002–2020 SAT benchmark instances.

Table A2. Mean (standard deviation) of the normalized values of Boolean structural measures on 2002–2020 SAT benchmark instances.

	Industrial	Crafted	Random
Backbone size	0.13 (0.2)	0.26 (0.39)	0.06 (0.23)
Backbone frequency	0.05% (0.18)	0.34% (1.14)	1.09% (3.63)
Backbone coverage	15.22% (23.92)	28.49% (40.33)	6.71% (23.74)
Backdoor size	0.67 (0.31)	0.88 (0.2)	0.98 (0.08)
Backdoor coverage	83.41% (25.1)	95.18% (12.73)	99.61% (3.49)
Backbone/backdoor variable overlap	0.08 (0.1)	0.29 (0.36)	0.22 (0.4)

Appendix B. Correlation and Exploitation Analysis

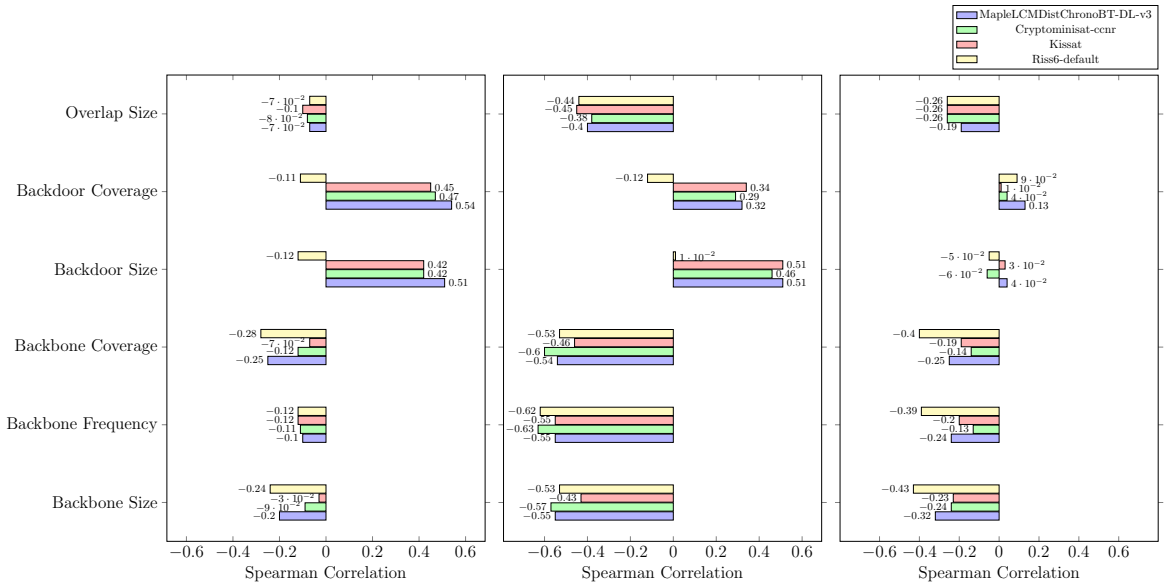
Appendix B.1. Correlation Analysis

Correlation analysis was carried out to quantify the strength and direction of the relation between the Boolean structural measures and the CDCL metrics. In particular, we computed the Spearman correlation coefficient to measure the nonlinear dependency between two random variables: the Boolean structural measures and CDCL metrics. Four CDCL solver metrics were evaluated: runtime, number of decisions, number of propagations, and number of conflicts. These metrics are derived from the essential blocks that make up any CDCL solver. The CDCL-based solvers that were examined are MapleLCMDistChronoBT-DL-v3, CryptoMiniSat-ccnr, Kissat, and Riss6-default. In choosing these solvers, we primarily considered their superior performance in SAT competitions and the unique differences in techniques incorporated and implementation.

Experiments were conducted on Shaheen II [48] and carried out on SAT instances presented in Table A1. A timeout period of 72 h was set to compute the four metrics runtime, number of decisions, number of propagation, and number of conflicts required

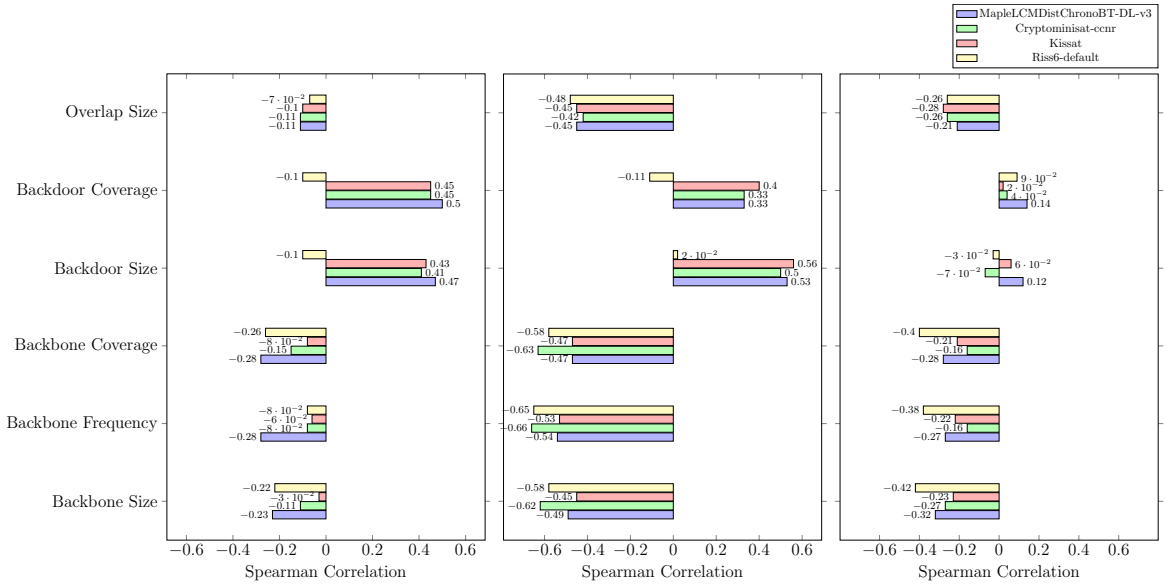
by the solvers: MapleLCMDistChronoBT-DL-v3, CryptoMiniSat-ccnr, Kissat, and Riss6-default for each of the 7779 SAT instances. As for the backbone and backdoor measures, they were computed as specified in Appendix A.

Results are depicted in Figures A4–A7. The results show that CDCL metrics are inversely proportional to backbone-related measures and backbone/backdoor overlap size, but they are proportional to backdoor-related measures across all industrial, crafted, and random benchmark instances. In addition, correlation results are apparent for structured instances. Finally, correlation results are more evident for MapleLCMDistChronoBT-DL-v3 on industrial benchmarks. Based on this, MapleLCMDistChronoBT-DL-v3 was selected as the base solver in our research, along with LSTech, which is an improvement of MapleLCMDistChronoBT-DL-v3 and a medal winner of the 2021 SAT competition.



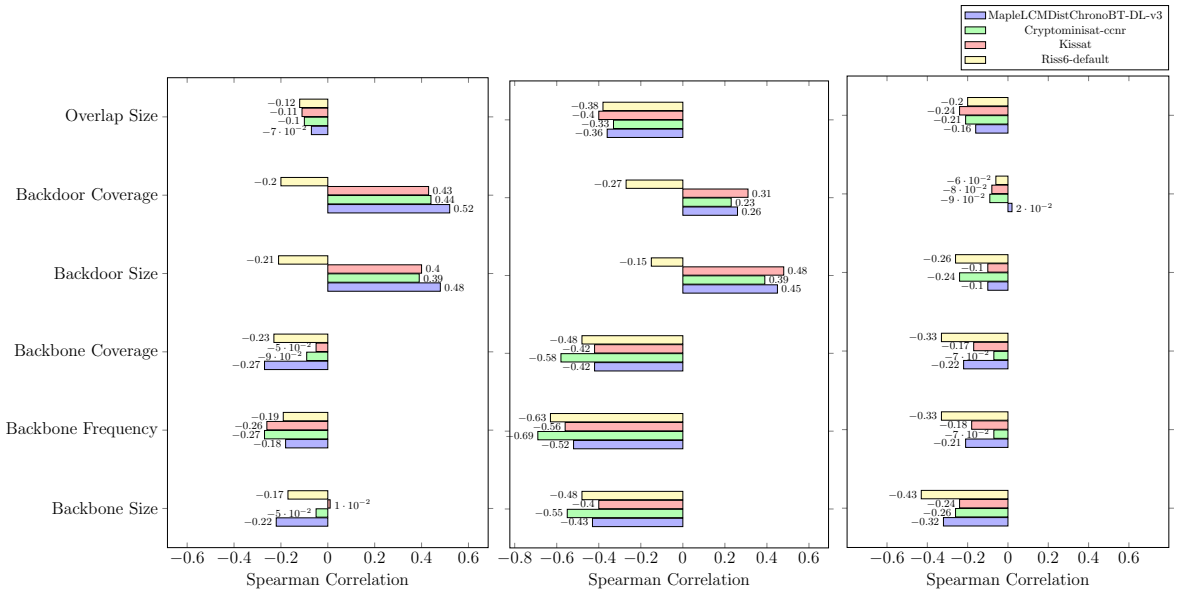
(**a**) Industrial benchmarks (**b**) Crafted benchmarks (**c**) Random benchmarks

Figure A4. Spearman correlation results between Boolean structural measures and the CPU time on 2002–2020 SAT benchmark instances.



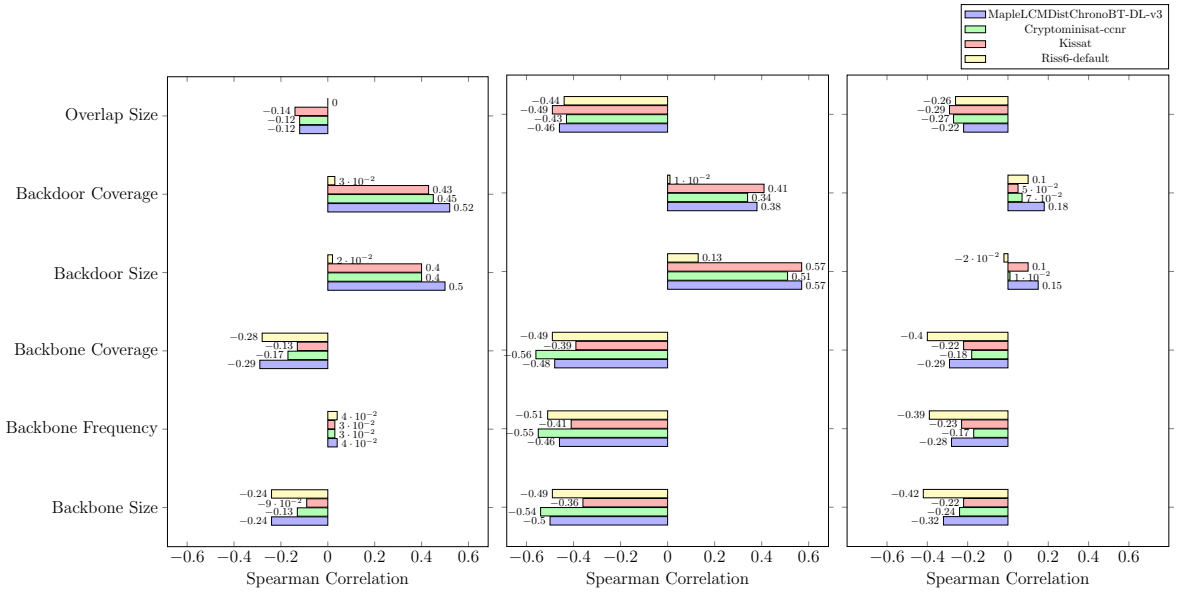
(**a**) Industrial benchmarks (**b**) Crafted benchmarks (**c**) Random benchmarks

Figure A5. Spearman correlation results between Boolean structural measures and the number of decisions on 2002–2020 SAT benchmark instances.



(**a**) Industrial benchmarks (**b**) Crafted benchmarks (**c**) Random benchmarks

Figure A6. Spearman correlation results between Boolean structural measures and the number of propagations on 2002–2020 SAT benchmark instances.



(**a**) Industrial benchmarks (**b**) Crafted benchmarks (**c**) Random benchmarks

Figure A7. Spearman correlation results between Boolean structural measures and the number of conflicts on 2002–2020 SAT benchmark instances.

Appendix B.2. Exploitation Analysis

Following the correlation results described in Appendix B.1, MapleLCMDistChronoBT-DL-v3 was used to carry out the rest of the analysis and experiments. As a way of examining whether MapleLCMDistChronoBT-DL-v3 exploits Boolean structural measures, we determined the percentage number of backbone variables decided for the solver in the first 1000 iterations. The experiments were limited to only backbone variables, not backdoors, since backbones are unique in SAT instances while backdoors are not. Therefore, determining whether backdoor variables are exploited is challenging.

Experiments were conducted, as before, on Shaheen II [48] and carried out on satisfiable SAT instances with backbones computed from 2002–2020 SAT benchmark instances (see Figure A1). The decision variables for the first 1000 decisions ran by MapleLCMDistChronoBT-DL-v3 were collected.

The percentage of exploited backbones performed by MapleLCMDistChronoBT-DL-v3 for the first 1000 decisions on all 2002–2020 benchmark instances is displayed in Figure A8. Generally, it seems that the percentage of exploited backbones varies more on industrial and crafted benchmark instances than on random ones. In addition, for structured instances, there appears to be a percentage number of instances where all backbone variables are exploited.

Furthermore, we analyzed the percentage of exploited backbones for SAT instances that have been solved in less than 1000 decisions for all benchmark categories. Each row in Tables A3–A5 (In the case of crafted instances, we grouped three instances per row for convenience.) identifies an instance, highlighting the number of decisions required to solve it (first column) and the percentage backbone variables exploited (second column). For instances that can be solved in less than 1000 decisions, it is obvious that structured instances exploit most of the backbone variables. On the other hand, less than 50% of backbone variables are exploited for random benchmark instances.

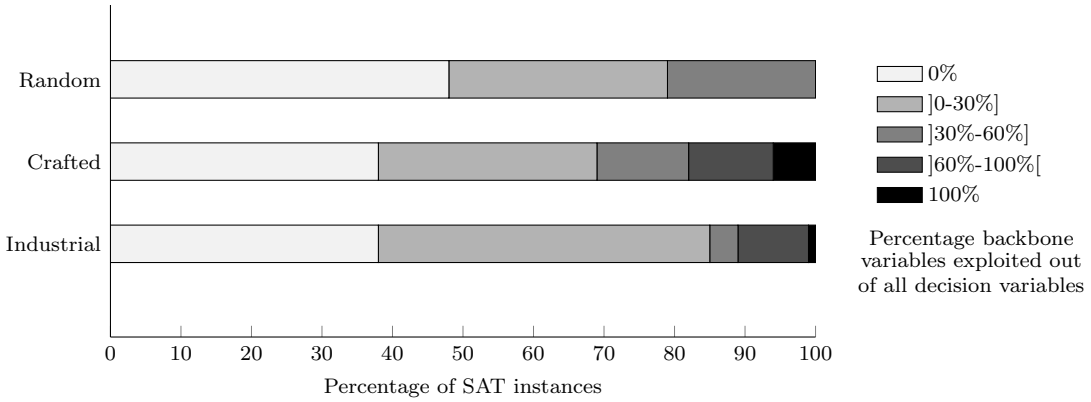


Figure A8. Percentage number of exploited backbones by MapleLCMDistChronoBT-DL-v3 for the first 1000 decisions on all 2002–2020 benchmark instances.

Table A3. Reported industrial instances from experiment in Figure A8 that are solved in fewer than 1000 decisions.

Number of Decisions	Percentage of Exploited Backbones in All Decision Variables
370	54.05%
15	66.67%
180	70.00%
372	91.67%
335	95.52%
174	99.43%
3	100.00%
36	100.00%
49	100.00%
55	100.00%
57	100.00%
57	100.00%
60	100.00%
152	100.00%

Table A4. Reported crafted instances from experiment in Figure A8 that are solved in fewer than 1000 decisions.

Number of Decisions	Percentage of Exploited Backbones in All Decision Variables	Number of Decisions	Percentage of Exploited Backbones in All Decision Variables	Number of Decisions	Percentage of Exploited Backbones in All Decision Variables
18	55.56%	44	86.36%	116	100.00%
19	57.89%	369	92.68%	144	100.00%
940	58.62%	184	98.37%	188	100.00%
563	58.79%	258	98.84%	208	100.00%
352	61.08%	215	99.07%	229	100.00%
443	61.63%	217	99.08%	239	100.00%
716	62.15%	224	99.11%	276	100.00%
534	63.11%	173	99.42%	281	100.00%
789	63.88%	16	100.00%	298	100.00%
490	65.51%	26	100.00%	300	100.00%
160	67.50%	30	100.00%	323	100.00%
61	70.49%	88	100.00%	346	100.00%
22	72.73%	99	100.00%	351	100.00%
22	81.82%	106	100.00%	401	100.00%
33	81.82%	109	100.00%	435	100.00%

Table A5. Reported random instances from experiment in Figure A8 that are solved in fewer than 1000 decisions.

Number of Decisions	Percentage of Exploited Backbones in All Decision Variables
69	47.83%
169	47.93%
678	48.82%
352	49.72%
667	49.93%
191	50.79%
391	51.15%
156	51.28%
594	53.20%
656	53.20%
312	53.53%
404	54.21%

Appendix C. The CDCL Framework and Popular Variable Decision Heuristics

Appendix C.1. The CDCL Framework

The CDCL framework is depicted in Algorithm A1. After constraint propagation detects a conflict (lines 3–4), CDCL learns at least one reason for that conflict in the form of a new clause (lines 5 and 6), called learnt clause, which is added to the CNF formula. In this case, the CDCL SAT solver directly backtracks to an earlier level (lines 9–12). The instance is declared unsatisfiable if the backtracked level is zero. In the absence of a conflict, a branching variable is picked and the level of that variable is updated accordingly (lines 18–21). Once all variables are assigned, the formula is satisfied (line 2). In the event that the restart condition is triggered, the SAT solver will restart (lines 13–16).

Algorithm A1: CDCL framework.

```
   Input: φ: CNF formula with variables x ∈ X
   Output: satisfiable and solution, otherwise unsatisfiable
1  INITIALIZE()
2  while notALLVARIABLESASSIGNED() do
3      (φ, status) ←BOOLEANCONSTRAINTPROPAGATION(φ,solution)
4      if status is conflict then
5          (blevel, learntClauseVars, conflictSideVars)=CONFLICTANALYSIS()    // Clause learning
6          AFTERCONFLICTANALYSIS(learntClauseVars, conflictSideVars)
7          if blevel is equal to 0 then
8              return unsatisfiable
9          else
10             solution ← BACKTRACK(blevel,solution)                          // Backtrack
11             decisionLevel ← blevel
12             ONUNASSIGN(x) // Called when variable x is unassigned by backtracking or
                restart
13     else if restart condition is triggered then
14         solution ← BACKTRACK(0,solution)                                   // Restart
15         decisionLevel ← 0
16         ONUNASSIGN(x)     // Called when variable x is unassigned by backtracking or
            restart
17     else
18         x ←PICKBRANCHINGVARIABLE()                                         // Branch
19         ONASSIGN(x)
20         decisionLevel ← decisionLevel + 1
21         solution ← solution ∪ (x, value)
22 return (satisfiable,solution)
```

Appendix C.2. The VSIDS Heuristic

VSIDS refers to a class of branching heuristics commonly used in CDCL SAT solvers that rank all the variables in an SAT instance during the run of the solver. VSIDS is significantly more effective than other well-known heuristics, particularly when solving unsatisfiable SAT instances [51,54,55]. The VSIDS heuristic was originally proposed as part of the Chaff solver [6]. Since then, many variants have been proposed, including VSIDS implemented in MiniSat and the variable move-to-front (VMTF) decision heuristic. In general, VSIDS is characterized by additive bumping, multiplicative decay, and low computation overhead. This work will focus on the VSIDS MiniSat variant implemented in Maple-based series SAT solvers [56].

The policy of VSIDS implemented in Maple-based series SAT solvers [56] is presented in Algorithm A2. All procedures are called as part of the CDCL framework in Algorithm A1. In the initialization procedure (line 1), each variable has a floating point number, called activity, which is initialized to 0 (lines 4–7). Following a conflict analysis phase (line 9), the activities of all variables that led to the learnt clause (including those in the learnt clause) are additively bumped (increased), typically by 1, if their decision levels of variables are greater than the backtrack level, such variables receive more bumps to activity scores (line 14); otherwise, they are bumped by 0.5 (line 16). In addition, all activities of variables are decremented by multiplying them by a constant $0 < Decay < 1$ called the multiplicative decay factor (lines 18–20). To select a variable, the (unassigned) variable and polarity with the highest activity are chosen at each decision (line 24).

Algorithm A2: VSIDS decision heuristic. All procedures are part of the CDCL framework in Algorithm A1.

```
// Called once at the start of the solver.
Procedure INITIALIZE()
    ...
    for x ∈ X                                   // X is the set of Boolean variables in φ.
    do
                                                // Initialize the activity of each variable to be 0.
        activity_x ← 0
    ...
// Called after a learnt clause is generated from conflict analysis.
Procedure AFTERCONFLICTANALYSIS(conflictSideVars ⊆ X , learntClauseVars ⊆ X)
    ...
    for x ∈ (conflictSideVars ∪ learntClauseVars)
    do
        if x_decisionLevel > backtrackLevel then
            bumpActivity(x, 1)   // Bump: increase the activities of all variables that
              led to the learnt clause including variables in the learnt clause by 1
              if their decision levels are greater than the backtrack level.
        else
            bumpActivity(x, 0.5) // Bump: increase the activities of all variables that
              led to the learnt clause including variables in the learnt clause by 0.5
              if their decision levels are less than the backtrack level.
    for x ∈ X do
        decayActivity(x, Decay)          // Decay: Multiply the activities of every variable by
          0 < Decay < 1.
        ...
// Called when the solver requests the next branching variable.
Procedure PICKBRANCHINGVARIABLE()
    ...
    return Unassigned x with highest activity_x
```

Appendix D. Reported Results on Industrial Families

Table A6. The number of solved instances of industrial families from the 2002–2021 SAT competitions. Each family consists of seven satisfiable instances.

Family	MapleLCM DistChrono BT-DL-v3	Maple_ VBBsfr_v1	LSTech	LSTech_ VBBsfrc_v2	Relaxed_ LCMDCBDL_ newTech	Kissat_GB	Kissat_MAB
aloul_bart	7	7	7	7	7	7	7
goldberg	7	7	7	7	7	7	7
grastien_anbulagan_diag	7	7	7	7	7	7	7
grastien_anbulagan_medium	7	7	7	7	7	7	7
rintanen	5	6	5	7	6	4	5
soos	5	6	5	7	6	5	5
Argumentation	7	7	7	7	7	6	7
StedmanTriples	7	7	7	7	6	6	7
MinimalSuperpermutation	6	7	7	7	6	6	7
biere_dinphil	7	7	7	7	7	7	7
eichberger	7	7	7	7	7	7	7
manthey_cvc4	6	5	7	6	6	7	7
corblin	7	7	7	7	7	7	7
cryptanalysis_zaikin	7	7	7	7	7	7	7
surynek	7	7	7	7	7	7	7
stojadinovic	7	7	7	7	7	7	7
zarpas_Ibm	7	7	7	7	7	7	7
manthey_stp	7	7	7	7	7	7	7
HamiltonianCycle	7	7	4	5	4	7	7
fuhs_aes	5	4	6	6	6	6	7
MaxsatOptimum	7	7	7	7	7	7	7

Table A6. *Cont.*

Family	MapleLCM DistChrono BT-DL-v3	Maple_ VBBsfr_v1	LSTech	LSTech_ VBBsfrc_v2	Relaxed_ LCMDCBDL_ newTech	Kissat_GB	Kissat_MAB
CircuitMultiplie	6	5	7	7	6	6	6
GiraldezCr	7	7	7	7	7	7	7
scheduling	7	7	7	7	7	1	7
PetrinetConcurrency	5	7	7	7	7	3	7
velve	7	7	7	7	7	7	7
ehlers	4	4	5	5	6	4	6
Total	175	177	179	183	178	166	183

Table A7. The number of decided unsatisfiable instances of industrial families from the 2002–2021 SAT competitions. Each family consists of seven unsatisfiable instances.

Family	MapleLCM DistChrono BT-DL-v3	Maple_ VBBsfr_v1	LSTech	LSTech_ VBBsfrc_v2	Relaxed_ LCMDCBDL_ newTech	Kissat_GB	Kissat_MAB
rintanen	6	7	7	7	7	6	6
eichberger	7	7	7	7	7	7	7
stojadinovic	7	7	7	7	7	7	7
surynek	7	7	7	7	7	7	7
GiraldezCr	6	7	6	6	6	6	6
manthey_cvc4	4	5	5	4	5	5	4
manthey_stp	5	5	5	5	5	5	5
ehlers	6	6	6	6	6	6	6
zaikin	7	7	7	7	7	7	7
goldberg	5	6	6	7	6	7	6
biere_dinphil	7	7	7	7	7	7	7
velve	7	7	7	7	7	7	7
zarpas_Ibm	7	7	7	7	7	7	7
corblin	7	7	7	7	7	7	7
AtLeastTwoSolutions	7	7	7	7	7	7	7
CellularAutomata	7	7	7	7	7	7	7
strcmpVerification	7	7	7	7	7	7	7
lamProblem	6	6	6	6	6	6	6
populationSafety	7	7	7	7	7	7	6
fuhs_aes	5	4	4	4	3	6	6
Total	127	130	129	129	128	131	128

References

1. Cook, S.A. The Complexity of Theorem-proving Procedures. In Proceedings of the Third Annual ACM Symposium on Theory of Computing, Shaker Heights, OH, USA, 3–5 May 1971; pp. 151–158. [CrossRef]
2. Lai, K.; Siewiorek, D.P. Functional testing of digital systems. In Proceedings of the 20th Design Automation Conference, DAC '83, Miami Beach, FL, USA, 27–29 June 1983; pp. 207–213.
3. Clarke, E.M.; Emerson, E.A. Design and synthesis of synchronization skeletons using branching time temporal logic. In Proceedings of the Logics of Programs, Yorktown Heights, NY, USA, 1982; Kozen, D., Ed.; Springer: Berlin/Heidelberg, Germany, 1982; pp. 52–71.
4. D'Silva, V.; Kroening, D.; Weissenbacher, G. A Survey of Automated Techniques for Formal Software Verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2008**, *27*, 1165–1178. [CrossRef]
5. SAT Competitions. 2002. Available online: http://www.satcompetition.org (accessed on 19 November 2019) .
6. Moskewicz, M.W.; Madigan, C.F.; Zhao, Y.; Zhang, L.; Malik, S. Chaff: Engineering an efficient SAT solver. In Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232), Las Vegas, NV, USA, 22 June 2001; pp. 530–535. [CrossRef]
7. Luby, M.; Sinclair, A.; Zuckerman, D. Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.* **1993**, *47*, 173–180. [CrossRef]
8. Audemard, G.; Simon, L. Predicting Learnt Clauses Quality in Modern SAT Solvers. In Proceedings of the 21st International Jont Conference on Artifical Intelligence, Pasadena, CA, USA, 11–17 July 2009; IJCAI'09; pp. 399–404.
9. Luo, M.; Li, C.M.; Xiao, F.; Manyà, F.; Lü, Z. An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, Melbourne, Australia 19–25 August 2017; pp. 703–711. [CrossRef]
10. Xiao, F.; Luo, M.; Li, C.M.; Manyà, F.; Lü, Z. MapleLRB LCM, Maple LCM, Maple LCM Dist, MapleLRB LCMoccRestart and Glucose-3.0+width in SAT Competition 2017. In Proceedings of the SAT Competition 2017: Solver and Benchmark Descriptions, Melbourne, Australia, 28 August–1 September 2017; Volume B-2017-1, pp. 25–26.

11. Nadel, A.; Ryvchin, V. Chronological Backtracking. In Proceedings of the Theory and Applications of Satisfiability Testing—SAT 2018, Oxford, UK, 9–12 July 2018; Beyersdorff, O., Wintersteiger, C.M., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 111–121.
12. Kochemazov, S.; Zaikin, O.; Semenov, A.A.; Kondratiev, V. Speeding Up CDCL Inference with Duplicate Learnt Clauses. In Proceedings of the ECAI 2020—24th European Conference on Artificial Intelligence, Santiago de Compostela, Spain, 29 August–8 September 2020; Giacomo, G.D., Catalá, A., Dilkina, B., Milano, M., Barro, S., Bugarín, A., Lang, J., Eds.; IOS Press: Shepherdsville, KY, USA, 2020; Volume 325, pp. 339–346. [CrossRef]
13. Liang, J.H.; Ganesh, V.; Poupart, P.; Czarnecki, K. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016; AAAI'16; pp. 3434–3440.
14. Liang, J.H.; Ganesh, V.; Poupart, P.; Czarnecki, K. Learning Rate Based Branching Heuristic for SAT Solvers. In Proceedings of the Theory and Applications of Satisfiability Testing—SAT 2016—19th International Conference, Bordeaux, France, 5–8 July 2016; Creignou, N., Berre, D.L., Eds.; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9710, pp. 123–140. [CrossRef]
15. Zhang, X.; Cai, S. Relaxed Backtracking with Rephasing. In Proceedings of the SAT Competition 2020, Alghero, Italy, 3–10 July 2020; Solver and Benchmark Descriptions; University of Helsinki, Department of Computer Science: Helsinki, Finland, 2020, Volume B-2020-1, pp. 15–16.
16. Cheeseman, P.; Kanefsky, B.; Taylor, W.M. Where the Really Hard Problems Are. In Proceedings of the 12th International Joint Conference on Artificial Intelligence—Volume 1, Sydney, Australia, 24–30 August 1991; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1991; IJCAI'91; pp. 331–337.
17. Monasson, R.; Zecchina, R.; Kirkpatrick, S.; Selman, B.; Troyansky, L. Determining computational complexity from characteristic 'phase transitions'. *Nature* **1999**, *400*, 133–137. [CrossRef]
18. Williams, R.; Gomes, C.P.; Selman, B. Backdoors to Typical Case Complexity. In Proceedings of the 18th International Joint Conference on Artificial Intelligence, Acapulco, Mexico, 9–15 August 2003; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2003; IJCAI'03; pp. 1173–1178.
19. Zulkoski, E.; Martins, R.; Wintersteiger, C.M.; Robere, R.; Liang, J.H.; Czarnecki, K.; Ganesh, V. Learning-Sensitive Backdoors with Restarts. In Proceedings of the Principles and Practice of Constraint Programming 2018, Lille, France, 27–31 August 2018; Volume 11008, pp. 453–469.
20. Kilby, P.; Slaney, J.; Thiébaux, S.; Walsh, T. Backbones and Backdoors in Satisfiability. In Proceedings of the 20th National Conference on Artificial Intelligence, Pittsburgh, PA, USA, 9–13 July 2005; AAAI'05; pp. 1368–1373.
21. Gregory, P.; Fox, M.; Long, D. A New Empirical Study of Weak Backdoors. In Proceedings of the Principles and Practice of Constraint Programming, Sydney, Australia, 14–18 September 2008; Stuckey, P.J., Ed.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 618–623.
22. Paris, L.; Ostrowski, R.; Siegel, P.; Sais, L. Computing Horn Strong Backdoor Sets Thanks to Local Search. In Proceedings of the 2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06), Arlington, VA, USA, 13–15 November 2006; pp. 139–143. [CrossRef]
23. Kilby, P.; Slaney, J.; Walsh, T. The Backbone of the Travelling Salesperson. In Proceedings of the 19th International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, 30 July–5 August 2005; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2005; IJCAI'05; p. 175–180.
24. Dequen, G.; Dubois, O. kcnfs: An Efficient Solver for Random k-SAT Formulae. In Proceedings of the Theory and Applications of Satisfiability Testing, 6th International Conference, Santa Margherita Ligure, Italy, 5–8 May 2003; Volume 2919, pp.486–501. [CrossRef]
25. Alyahya, T.N.; Menai, M.; Mathkour, H. On the Structure of the Boolean Satisfiability Problem: A Survey. *ACM Comput. Surv.* **2022**, *55*, 1–34.
26. Janota, M.; Lynce, I.; Marques-Silva, J. Algorithms for computing backbones of propositional formulae. *AI Commun.* **2015**, *28*, 161–177. [CrossRef]
27. Dilkina, B.N.; Gomes, C.P.; Sabharwal, A. Backdoors in the Context of Learning. In Proceedings of the Theory and Applications of Satisfiability Testing 2009, Swansea, Wales, UK, 30 June–3 July 2009; Volume 5584, pp. 73–79.
28. Davis, M.; Putnam, H. A Computing Procedure for Quantification Theory. *J. ACM* **1960**, *7*, 201–215. [CrossRef]
29. Marques-Silva, J.P.; Sakallah, K.A. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.* **1999**, *48*, 506–521. [CrossRef]
30. Goldberg, E.; Novikov, Y. BerkMin: A fast and robust Sat-solver. *Discret. Appl. Math.* **2007**, *155*, 1549–1561. [CrossRef]
31. Gomes, C.P.; Selman, B.; Kautz, H.A. Boosting Combinatorial Search Through Randomization. In Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, Madison, WI, USA, 26–30 July 1998; pp. 431–437.
32. Eén, N.; Sörensson, N. An Extensible SAT-solver. In Proceedings of the Theory and Applications of Satisfiability Testing 2003, Santa Margherita Ligure, Italy, 5–8 May 2003; Volume 2919, pp. 502–518.
33. Soos, M.; Nohl, K.; Castelluccia, C. Extending SAT Solvers to Cryptographic Problems. In Proceedings of the Theory and Applications of Satisfiability Testing 2009, Swansea, Wales, UK, 30 June–3 July 2009; Volume 5584, pp. 244–257.

34. Biere, A.; Fazekas, K.; Fleury, M.; Heisinger, M. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Proceedings of the SAT Competition 2020: Solver and Benchmark Descriptions, Alghero, Italy, 5–9 July 2020; University of Helsinki, Department of Computer Science: Helsinki, Finland, 2020; Volume B-2020-1, pp. 50–53.
35. Oh, C. Between SAT and UNSAT: The Fundamental Difference in CDCL SAT. In Proceedings of the Theory and Applications of Satisfiability Testing—SAT 2015, Austin, TX, USA, 24–27 September 2015; pp. 307–323.
36. Ryvchin, V.; Nadel, A. Maple_LCM_Dist_ChronoBT: Featuring chronological backtracking. In Proceedings of the SAT Competition 2018: Solver and Benchmark Descriptions, Oxford, UK, 9–12 July 2018; Volume B-2028-1, p. 29.
37. Zhang, X.; Cai, S.; Zhihan, C. Improving CDCL via Local Search. In Proceedings of the SAT Competition 2021: Solver and Benchmark Descriptions, Barcelona, Spain, 5–9 July 2021; Volume B-2021-1, p. 42.
38. Solimul Chowdhury, M.; Müller, M.; You, J.H. Four CDCL solvers based on expLRB, expVSIDS and Glue Bumping. In Proceedings of the SAT Competition 2021: Solver and Benchmark Descriptions, Barcelona, Spain, 5–9 July 2021; Volume B-2021-1, pp. 17–18.
39. Cherif, M.S.; Habet, D.; Terrioux, C. Combining VSIDS and CHB Using Restarts in SAT. In Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP 2021), online, 25–29 October 2021; Volume 210, pp. 20:1–20:19. [CrossRef]
40. Kaiser, A.; Küchlin, W. *Detecting Inadmissible and Necessary Variables in Large Propositional Formulae*; Technical Report; University of Siena: Siena, Italy, 2001.
41. Climer, S.; Zhang, W. Searching for Backbones and Fat: A Limit-Crossing Approach with Applications. In Proceedings of the Eighteenth National Conference on Artificial Intelligence, Edmonton, Alberta, Canada, 28 July–1 August 2002; pp. 707–712.
42. Previti, A.; Järvisalo, M. A Preference-Based Approach to Backbone Computation with Application to Argumentation. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing, Pau, France, 9–13 April 2018; Association for Computing Machinery: New York, NY, USA, 2018; SAC '18; pp. 896–902. [CrossRef]
43. Zhang, Y.; Zhang, M.; Pu, G. Optimizing Backbone Filtering. *Sci. Comput. Program.* **2020**, *187*, 102374. [CrossRef]
44. Wu, H. Improving SAT-Solving with Machine Learning. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, WA, USA, 8–11 March 2017; Association for Computing Machinery: New York, NY, USA, 2017; SIGCSE '17; pp. 787–788. [CrossRef]
45. Dilkina, B.N.; Gomes, C.P.; Malitsky, Y.; Sabharwal, A.; Sellmann, M. Backdoors to Combinatorial Optimization: Feasibility and Optimality. In Proceedings of the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009, Pittsburgh, PA, USA, 27–31 May 2009; pp. 56–70. [CrossRef]
46. Achterberg, T.; Koch, T.; Martin, A. MIPLIB 2003. *Oper. Res. Lett.* **2006**, *34*, 361–372. [CrossRef]
47. Menai, M.E.; Batouche, M. A Backbone-Based Co-evolutionary Heuristic for Partial MAX-SAT. In Proceedings of the Artificial Evolution, 7th International Conference, Evolution Artificielle, EA 2005, Lille, France, 26–28 October 2005; Revised Selected Papers; Talbi, E., Liardet, P., Collet, P., Lutton, E., Schoenauer, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3871, pp. 155–166. [CrossRef]
48. Hadri, B.; Kortas, S.; Feki, S.; Khurram, R.; Newby, G. Overview of the KAUST's Cray X40 system–Shaheen II. In Proceedings of the 2015 Cray User Group, Chicago, IL, USA, 26–30 April 2015.
49. Williams, R.; Gomes, C.; Selman, B. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. *Structure* **2003**, *23*, 4.
50. Zulkoski, E.; Martins, R.; Wintersteiger, C.M.; Robere, R.; Liang, J.; Czarnecki, K.; Ganesh, V. Relating Complexity-theoretic Parameters with SAT Solver Performance. *arXiv* **2017**, arXiv:1706.08611.
51. Kochemazov, S. Improving Implementation of SAT Competitions 2017-2019 Winners. In Proceedings of the Theory and Applications of Satisfiability Testing—SAT 2020—23rd International Conference, Alghero, Italy, 3–10 July 2020; Volume 12178, pp. 139–148. [CrossRef]
52. Gomes, C.P.; Selman, B.; Crato, N.; Kautz, H.A. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *J. Autom. Reason.* **2000**, *24*, 67–100. [CrossRef]
53. Zulkoski, E.; Martins, R.; Wintersteiger, C.M.; Liang, J.H.; Czarnecki, K.; Ganesh, V. The Effect of Structural Measures and Merges on SAT Solver Performance. In Proceedings of the Principles and Practice of Constraint Programming 2018, Lille, France, 27–37 August 2018; Volume 11008, pp. 436–452.
54. Liang, J.H.; Ganesh, V.; Zulkoski, E.; Zaman, A.; Czarnecki, K. Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers. In Proceedings of the Haifa Verification Conference, Haifa, Israel, 17–19 November 2015; Volume 9434, pp. 225–241.
55. Luo, M.; Li, C.; Wu, X.; Li, S.; Lü, Z. Branching Strategy Selection Approach Based on Vivification Ratio. *arXiv* **2021**, arXiv:2112.06917.
56. Liang, J.H.; Oh, C.; Ganesh, V.; Czarnecki, K. MapleCOMSPS, MapleCOMSPS LRB, MapleCOMSPS CHB. In Proceedings of the SAT Competition 2016: Solver and Benchmark Descriptions, Bordeaux, France, 5–8 July 2016; Volume B-2016-1, pp. 52–53.