Designing New Phase Selection Heuristics

Arijit Shaw and Kuldeep S. Meel

School of Computing, National University of Singapore

Abstract. CDCL-based SAT solvers have transformed the field of automated reasoning owing to their demonstrated efficiency at handling problems arising from diverse domains. The success of CDCL solvers is owed to the design of clever heuristics that enable the tight coupling of different components. One of the core components is phase selection, wherein the solver, during branching, decides the polarity of the branch to be explored for a given variable. Most of the state-of-the-art CDCL SAT solvers employ *phase-saving* as a phase selection heuristic, which was proposed to address the potential inefficiencies arising from *far-backtracking*. In light of the emergence of chronological backtracking in CDCL solvers, we re-examine the efficiency of phase saving. Our empirical evaluation leads to a surprising conclusion: The usage of saved phase and random selection of polarity for decisions following a chronological backtracking leads to an indistinguishable runtime performance in terms of instances solved and PAR-2 score.
We introduce Decaying Polarity Score (DPS) to capture the *trend* of the polarities attained by the variable, and upon observing lack of performance improvement due to DPS, we turn to a more sophisticated heuristic seeking to capture the activity of literals and the trend of polarities: Literal State Independent Decaying Sum (LSIDS). We find the 2019 winning SAT solver, Maple_LCM_Dist_ChronoBTv3, augmented with LSIDS solves 6 more instances while achieving a reduction of over 125 seconds in PAR-2 score, a significant improvement in the context of the SAT competition.

1 Introduction

Given a Boolean formula F, the problem of Boolean Satisfiability (SAT) asks whether there exists an assignment σ such that σ satisfies F. SAT is a fundamental problem in computer science with wide-ranging applications including bioinformatics [24], AI planning [18], hardware and system verification [7,9], spectrum allocation, and the like. The seminal work of Cook [10] showed that SAT is NP-complete and the earliest algorithmic methods, mainly based on local search and the DPLL paradigm [11], suffered from scalability in practice. The arrival of Conflict Driven Clause Learning (CDCL) in the early '90s [36] ushered in an era of sustained interest from theoreticians and practitioners leading to a medley of efficient heuristics that have allowed SAT solvers to scale to instances involving millions of variables [25], a phenomenon often referred to as *SAT revolution* [2,6,12,22,26,28,29,36].

The progress in modern CDCL SAT solving over the past two decades owes to the design and the tight integration of the core components: *branching* [19,35], *phase selection* [32], *clause learning* [3,23], *restarts* [4,14,16,21], and *memory management* [5,31]. The progress has often been driven by the improvement of the state of the art heuristics for the core components. The annual SAT competition [17] is witness to the pattern where development of the heuristics for one core component necessitates and encourages the design of new heuristics for other components to ensure a tight integration.

The past two years have witnessed the (re-)emergence of chronological backtracking, a regular feature of DPLL techniques, after almost a quarter-century since the introduction of non-chronological backtracking (NCB), thanks to Nadel and Ryvchin [29]. The impact of chronological backtracking (CB) heuristics is evident from its quick adoption by the community, and the CB-based solver, Maple_LCM_Dist _ChronoBT [33], winning the SAT Competition in 2018 and and a subsequent version, Maple_LCM_Dist _ChronoBTv3 , the SAT Race 2019 [15] winner. The 2nd best solver at the SAT Race 2019, CaDiCaL, also implements chronological backtracking. The emergence of chronological backtracking necessitates re-evaluation of the heuristics for the other components of SAT solving.

We turn to one of the core heuristics whose origin traces to the efforts to address the inefficiency arising due to loss of information caused by non-chronological backtracking: the *phase saving* [32] heuristic in the phase selection component. When the solver decides to branch on a variable v, the phase selection component seeks to identify the polarity of the branch to be explored by the solver. The idea of phase-saving traces back to the field of constraint satisfaction search [13] and SAT checkers [34], and was introduced in CDCL by Pipatsrisawat and Darwiche [32] in 2007. For a given variable v, phase saving returns the polarity of v corresponding to the last time v was assigned (via decision or propagation). The origin of phase saving traces to the observation by Pipatsrisawat and Darwiche that for several problems, the solver may forget a valid assignment to a subset of variables due to non-chronological backtracking and be forced to *re-discover* the earlier assignment. In this paper, we focus on the question: *is phase saving helpful for solvers that employ chronological backtracking? If not, can we design a new phase selection heuristic?*

The primary contribution of this work is a rigorous evaluation process to understand the efficacy of phase saving for all decisions following a chronological backtracking and subsequent design of improved phase selection heuristic. In particular,

1. We observe that in the context of 2019's winning SAT solver Maple_LCM_Dist _ChronoBTv3 (referred to as mldc henceforth)[1], phase saving heuristic for decisions following a chronological backtracking performs no better than the random heuristic which assigns positive or negative polarity randomly with probability 0.5.

[1] acronyms in sans serif font denote solvers or solvers with some specific configurations. mldc is used as abbreviation for Maple_LCM_Dist_ChronoBTv3

2. To address the inefficacy of phase saving for decisions following a chronological backtracking, we introduce a new metric, *decaying polarity score* (DPS), and DPS-based phase selection heuristic. DPS seeks to capture the *trend* of polarities assigned to variables with higher priority given to recent assignments. We observe that augmenting mldc with DPS leads to almost the same performance as the default mldc, which employs phase saving as the phase selection heuristic.
3. To meet the dearth of performance gain by DPS, we introduce a sophisticated variant of DPS called *Literal State Independent Decaying Sum*(LSIDS), which performs additive bumping and multiplicative decay. While LSIDS is inspired by VSIDS, there are crucial differences in computation of the corresponding activity of literals that contribute significantly to the performance. Based on empirical evaluation on SAT 2019 instances, mldc augmented with LSIDS, called mldc-lsids-phase solves 6 more instances and achieves the PAR-2 score of 4475 in comparison to 4607 seconds achieved by the default mldc.
4. To determine the generality of performance improvement of mldc-lsids-phase over mldc; we perform an extensive case study on the benchmarks arising from *preimage attack* on SHA-1 cryptographic hash function, a class of benchmarks that achieves significant interest from the security community.

The rest of the paper is organized as follows. We discuss background about the core components of the modern SAT solvers in Section 2. Section 3 presents an empirical study to understand the efficacy of phase saving for decisions following a chronological backtracking. We then present DPS-based phase selection heuristic and the corresponding empirical study in Section 4. Section 5 presents the LSIDS-based phase selection heuristic. We finally conclude in Section 6.

2 Background

A literal is a propositional variable v or its negation $\neg v$. A Boolean formula F over the set of variables V is in Conjunctive Normal Form (CNF) if F is expressed as conjunction of clauses wherein each clause is a disjunction over a subset of literals. A truth assignment $\sigma : V \mapsto \{0, 1\}$ maps every variable to 0 (False) or 1 (True). An assignment σ is called satisfying assignment or solution of F if $F(\sigma) = 1$. The problem of Boolean Satisfiability (SAT) seeks to ask whether there exists a satisfying assignment of F. Given F, a SAT solver is expected to return a satisfying assignment of F if there exists one, or a proof of unsatisfiability [37].

2.1 CDCL solver

The principal driving force behind the so-called *SAT revolution* has been the advent of the Conflict Driven Clause Learning (CDCL) paradigm introduced by Marques-Silva and Sakallah [36], which shares syntactic similarities with the DPLL paradigm [11] but is known to be exponentially more powerful in theory. The power of CDCL over DPLL is not just restricted to theory, and its practical impact is evident from the observation that all the winning SAT solvers in the main track have been CDCL since the inception of SAT competition [17] .

On a high-level, a CDCL-based solver proceeds with an empty set of assignments and at every time step maintains a *partial assignment*. The solver iteratively assigns a subset of variables until the current partial assignment is determined not to satisfy the current formula, and the solver then backtracks while learning the reason for the unsatisfiability expressed as a conflict clause. The modern solvers perform frequent restarts wherein the partial assignment is set to empty, but information from the run so far is often stored in form of different statistical metrics. We now provide a brief overview of the core components of a modern CDCL solver.

1. **Decision.** The decision component selects a variable v, called the *decision variable* from the set of unassigned variables and assigns a truth value, called the *decision phase* to it. Accordingly, a Decision heuristic is generally a combination of two different heuristics – a *branching heuristic* decides the decision variable and a *phase selection heuristic* selects the decision phase. A *decision level* is associated with each of the decision variables while it gets assigned. The count for decision level starts at 1 and keeps on incrementing with every decision.
2. **Propagation.** The propagation procedure computes the direct implication of the current partial assignment. For example, some clauses become *unit* (all but one of the literals are False) with the decisions recently made by the solver. The remaining unassigned literal of that clause is asserted and added to the partial assignment by the propagation procedure. All variables that get assigned as a consequence of the variable v get the same decision level as v.
3. **Conflict Analysis.** Propagation may also reveal that the formula is not satisfiable with the current partial assignment. The situation is called a *conflict*. The solver employs a *conflict analysis* subroutine to deduce the reason for unsatisfiability, expressed as a subset of the current partial assignment. Accordingly, the *conflict analysis* subroutine returns the negation of the literals from the subset as a clause c, called a *learnt clause* or *conflict clause* which is added to the list of the existing clauses. The clauses in the given CNF formula essentially imply the learnt clause.
4. **Backtrack.** In addition to leading a learnt clause, the solver then seeks to undo a subset of the current partial assignment. To this end, the conflict analysis subroutine computes the backtrack decision level l, and then the solver deletes assignment to all the variables with decision level greater than l. As the backtrack intimates removing assignment of last decision level only, backtracking for more than one level is also called *non-chronological backtracking* or, *backjumping*.

The solver keeps on repeating the procedures as mentioned above until it finds a satisfying assignment or finds a conflict without any assumption. The ability of modern CDCL SAT solvers to solve real-world problems with millions of variables depends on its highly sophisticated heuristics employed in different components of the solver. Now we discuss some terms related to CDCL SAT solving that we use extensively in the paper.

– *Variable state independent decaying sum* (VSIDS) introduced in Chaff [28] refers to a branching heuristic, where a score called *activity* is maintained for every variable. The variable with the highest *activity* is returned as the decision variable. Among different variations of VSIDS introduced later, the most effective is *Exponential VSIDS* or, EVSIDS [8,20] appeared in MiniSat [12]. The EVSIDS score for variable v, $activity[v]$, gets incremented additively by a factor f every time v appears in a learnt clause. The factor f itself also gets incremented multiplicatively after each conflict. A constant factor $g = 1/f$ periodically decrements the *activity* of all the variables. The act of increment is called *bump*, and the decrement is called *decay*. The heuristic is called *state independent* because the *activity* of a variable is not dependent of the current state (e.g., current assumptions) of the solver.
– *Phase saving* [32] is a phase selection heuristic used by almost all solver modern solvers, with few exceptions such as the CaDiCaL solver in SAT Race 19 [15]. Every time the solver backtracks and erases the current truth assignment, phase saving stores the erased assignment. For any variable, only the last erased assignment is stored, and the assignment replaces the older stored assignment. Whenever the branching heuristic chooses a variable v as the decision variable and asks phase saving for the decision phase, phase saving returns the saved assignment.
– *Chronological backtracking.* When a non-chronological solver faces a *conflict*, it *backtracks* for multiple levels. Nadel et al. [29] suggested non-chronological backtracking (NCB) might not always be helpful, and advocated backtracking to the previous decision level. The modified heuristic is called *chronological backtracking* (CB). We distinguish a decision based on whether the last backtrack was chronological or not. If the last backtrack is chronological, we say the solver is in *CB-state*, otherwise the solver is in *NCB-state*.

2.2 Experimental Setup

In this work, our methodology for the design of heuristics has focused on the implementation of heuristics on a base solver and conduction of an experimental evaluation on a high-performance cluster for SAT 2019 benchmarks. We now describe our experimental setup in detail. All the empirical evaluations in this paper used this setup, unless mentioned otherwise.

1. **Base Solver :** We implemented the proposed heuristics on top of the solver Maple_LCM_Dist_ChronoBTv3 (mldc), which is the winning solver for SAT Race 2019. Maple_LCM_Dist_ChronoBTv3 is an modification of Maple_LCM_Dist_ChronoBT (2018), which implements chronological backtracking on top of Maple_LCM_Dist (2017). Maple_LCM_Dist, in turn, evolved from MiniSat (2006) through Glucose (2009) and MapleSAT (2016). The

years in parenthesis represent the year when the corresponding solver was published.

2. **Code Blocks**: The writing style of this paper is heavily influenced from the presentation of MiniSat by Eén and Sörensson [12]. Following Eén and Sörensson, we seek to present implementation details as code blocks that are intuitive yet detailed enough to allow the reader to implement our heuristics in their own solver. Furthermore, we seek to present not only the final heuristic that performed the best, but we also attempt to present closely related alternatives and understand their performance.
3. **Benchmarks :** Our benchmark suite consisted of the entire suite, totaling 400 instances, from SAT Race '19.
4. **Experiments :** We conducted all our experiments on a high-performance computer cluster, with each node consists of E5-2690 v3 CPU with 24 cores and 96GB of RAM. We used 24 cores per node with memory limit set to 4GB per core, and all individual instances for each solver were executed on a single core. Following the timeout used in SAT competitions, we put a timeout of 5000 seconds for all experiments, if not otherwise mentioned. In contrast to SAT competition, the significant difference in specifications of the system lies in the size of RAM: our setup allows 4 GB of RAM in comparison to 128 GB of RAM allowed in SAT race '19.
   We computed the number of SAT and UNSAT instances the solver can solve with each of the heuristics. We also calculated the PAR-2 score. The PAR-2 score, an acronym for penalized average runtime, used in SAT competitions as a parameter to decide winners, assigns a runtime of two times the time limit (instead of a "not solved" status) for each benchmark not solved by the solver.[2]

3 Motivation

The impressive scalability of CDCL SAT solvers owes to the tight coupling among different components of the SAT solvers wherein the design of heuristic is influenced by its impact on other components. Consequently, the introduction of a new heuristic for one particular component requires one to analyze the efficacy of the existing heuristics in other components. To this end, we seek to examine the efficacy of phase saving in the context of recently introduced heuristic, Chronological Backtracking (CB). As mentioned in Section 1, the leading SAT solvers have incorporated CB and therefore, we seek to revisit the efficacy of other heuristics in light of CB. As a first step, we focus on the evaluation of phase selection heuristic.

Phase saving was introduced to tackle the loss of precious work due to *far-backtracking* [32]. Interestingly, CB was introduced as an alternative to *far-bactracking*, i.e., when the conflict analysis recommends that the solver should backtrack to a level $\hat{l}$ such that $|l - \hat{l}|$ is greater than a chosen threshold (say, *thresh*), CB instead leads the solver to backtrack to the previous level. It is

[2] All experimental data are available at `https://doi.org/10.5281/zenodo.3817476`.

worth noting that if the conflict analysis returns $\hat{l}$ such that $l - \hat{l} < thresh$, then the solver does backtrack to $\hat{l}$. Returning to CB, since the solver in CB-state does not perform far-backtracking, it is not clear if phase saving in CB-state is advantageous. To analyze empirically, we conducted preliminary experiments with mldc, varying the phase-selection heuristics while the solver is performing CB. We fix the phase selection to phase saving whenever the solver performs NCB and vary the different heuristics while the solver performs CB:

1. *Phase-saving* : Choose the saved phase as polarity, default in mldc.
2. *Opposite of saved phase* : Choose the negation of the saved phase for the variable as polarity.
3. *Always false*: The phase is always set to FALSE, a strategy that was originally employed in MiniSat 1.0.
4. *Random* : Randomly choose between FALSE and TRUE.

Our choice of *Random* among the four heuristics was driven by our perception that a phase selection strategy should be expected to perform better than *Random*. Furthermore, to put the empirical results in a broader context, we also experimented with the random strategy for both NCB and CB. The performance of different configurations is presented in Table 1, which shows a comparison in terms of the number of SAT, UNSAT instances solved, and PAR-2 score.

Phase selection heuristic used					
In NCB-state	In CB-state	SAT	UNSAT	Total	PAR-2
Random	*Random*	133	89	222	5040.59
Phase-saving	*Phase-saving*	140	97	237	4607.61
Phase-saving	*Random*	139	100	239	4537.65
Phase-saving	*Always false*	139	98	237	4597.06
Phase-saving	*Opp. of saved phase*	137	98	235	4649.13

Table 1: Performance of mldc on 400 SAT19 benchmarks while aided with different phase selection heuristics. SAT, UNSAT, and total columns indicate the number of SAT, UNSAT, and SAT+UNSAT instances solved by the solver when using the heuristic. A lower PAR2 score indicates a lower average runtime, therefore better performance of the solver.

We first observe that the mldc solves 237 instances and achieves a PAR-2 score of 4607 – a statistic that will be the baseline throughout the rest of the paper. Next, we observe that usage of random both in CB-state and NCB-state leads to significant degradation of performance: 15 fewer instances solved with an increase of 440 seconds for PAR-2. Surprisingly, we observe that random phase selection in CB-state while employing phase saving in NCB-state performs as good as phase-saving for CB-state. Even more surprisingly, we do not notice a significant performance decrease even when using *Always false* or *Opposite of* *saved phase*. These results strongly indicate that phase saving is not efficient when the solver is in CB-state, and motivate the need for a better heuristic. In the rest of the paper, we undertake the task of the searching for a better phase selection heuristic.

4 Decaying Polarity Score for Phase Selection

To address the ineffectiveness of phase saving in CB-state, we seek to design a new phase selection heuristic while the solver is in CB-state. As a first step, we view phase saving as remembering only the last assigned polarity and we intend to explore heuristic design based on the recent history of polarities assigned to the variable of interest. Informally, we would like to capture the weighted *trend* of the polarities assigned to the variable with higher weight to the recently assigned polarity. To this end, we maintain a score, represented as a floating-point number, for every variable and referred to as *decaying polarity score* (DPS). Each time the solver backtracks to level l, the assignments of all the variables with decision level higher than l are removed from the partial assignment. We update the respective *decaying polarity score* of all these variables, whose assignment gets canceled, using the following formula:

$$dps[v] = pol(v) + dec \times dps[v] \tag{1}$$

where,

- $dps[v]$ represent the *decaying polarity score* of the variable v.
- $pol(v)$ is $+1$ if polarity was set to True at the last assignment of the variable, -1 otherwise.
- The decay factor dec is chosen from $(0, 1)$. The greater the value of dec is, the more preference we put on polarities selected in older conflicts.

Whenever the branching heuristic picks a variable v to branch on, the DPS-based phase selection heuristic returns positive polarity if $dps[v]$ is positive; otherwise, negative polarity is returned. Note that for $dec \leq 0.5$, the DPS-based heuristic is equivalent to phase saving modulo the discrepancies arising due to differences in floating-point arithmetic and real number arithmetic.

4.1 Implementation

The implementation of DPS-based heuristic closely resembles the implementation of the *phase saving* heuristic [32]. Here we maintain an array of floating-point numbers named *dps*. Each of the elements in the array corresponds to the decaying polarity score of a variable in the formula. Whenever the solver backtracks and *erases* the assignment for a variable v, the $dps[v]$ gets updated using Equation 1. At any later point during solving, if the branching heuristic decides to branch on the variable v, the decaying polarity score heuristic returns a phase based on the value of $dps[v]$. The methods for updating *dps* and picking up a literal based on *dps* is shown in Figure 1.

```
void Solver.cancelUntil(int bLevel)        ▷ bLevel : backtrack level
 - for all elements which are getting cancelled
   for (int c = trail.size()−1; c >= trailLim[bLevel]; c−−)
      Var x = var(trail[c])
      dps[x] ∗= dec
      dps[x] += (sign(trail[c]) ? 1.0 : −1.0)
```

```
Lit pickBranchLit()
   next : variable returned by branching heuristic
   if (dps[next] > 0)
      lit = mkLit(next, true)
   else
      lit = mkLit(next, false)
   return lit
```

Fig. 1: Updating and using of decaying polarity score in MiniSat like code.

System	SAT	UNSAT	Total	PAR-2
mldc	140	97	237	4607.61
mldc-dec-phase-0.5	141	97	238	4589.66
mldc-dec-phase-0.7	141	96	237	4604.66

Table 2: Performance comparison of decaying polarity score with phase saving on SAT19 instances. mldc-dec-phase-<value> represent the the solvers using decaying polarity score. The <value> represent the *dec* used.

4.2 Experimental Results

To test the efficiency of DPS-based phase selection heuristic, we augmented[3] our base solver, mldc, with DPS-based phase selection heuristic during CB. We set the value of $dec = 0.5$ and 0.7 to understand the behavior with respect to varying values of *dec*. As discussed in subsection 2.2, we conducted our empirical evaluation on SAT-19 benchmarks.

Solved Instances and PAR-2 score Comparison. Table 2 presents the comparison of the number of instances solved and PAR-2 score. We first note that the usage of DPS did not result in a statistically significant change in the performance of mldc. Furthermore, the impact of *dec* seems fairly limited as well. In light of the above remark of equivalence of phase saving and DPC with $dec \leq 0.5$, the floating-point computations introduce non-determinism due to lack of arbitrary precision. Table 2 indicates that the usage of DPS does not seem to enhance the aggregate performance of mldc. It is worth noting that there are significantly many

[3] https://github.com/meelgroup/duriansat/tree/decay-pol

instances where mldc attains more than 20% improvement over mldc-dec-phase-0.7 and vice-versa. In 53 instances mldc-dec-phase-0.7 had 20% or more runtime improvement, while mldc had 20% or higher performance than mldc-dec-phase-0.7 in 70 instances. The interesting behavior demonstrated by heuristic indicates, while DPS-based phase selection heuristic fails to attain such an objective, it is possible to design heuristics that can accomplish performance improvement over phase saving. In the next section, we design a more sophisticated scheme that seeks to achieve the above goal.

5 LSIDS : A VSIDS like heuristic for phase selection

We now shift to a more sophisticated heuristic that attempts to not only remember the trend of activity but also aims to capture the *activity* of the corresponding literal. To this end, we introduce a scoring scheme, called Literal State Independent Decay Sum (LSIDS), that performs additive bumping and multiplicative decay, à la VSIDS and EVISDS style. The primary contribution lies in the construction of policies regarding literal bumping. We maintain activity for every literal, and the activity is updated as follows:

1. **Literal bumping** refers to incrementing *activity* for a literal. With every *bump*, the activity for a literal is incremented (additive) by $inc * mult$, where $mult$ is a fixed constant while at every conflict, inc gets multiplied by some factor $g > 1$. *Literal bumping* takes place in the following two different phases of the solver.
   Reason-based bumping
   When a clause c is learnt, for all the literals l_i appearing in c, the *activity* for l_i is bumped. For example, if we learn the clause that consists of literals v_5, $\neg v_6$ and v_3, then we bump the *activity* of literals v_5, $\neg v_6$ and v_3.
   Assignment-based bumping
   While an assignment for a variable v gets canceled during backtrack; if the assignment was True, then the solver bumps *activity* for v, otherwise the *activity* for $\neg v$ is bumped.
2. **Literal decaying** denotes the incident of multiplying the parameter inc by a factor > 1 at every conflict. The multiplication of inc implies the next bumps will be done by a higher inc. Therefore, the older bumps to *activity* will be relative smaller than the newer bumps. The name *decaying* underscores the fact that the effect of increasing inc is equivalent to decreasing (or, *decay*-ing) the *activity* of all literals.
3. **Literal rescoring** : As the *activity* gets incremented by a larger and larger factor every time, the value for *activity* reaches the limit of a floating-point number at some time. At this point the *activity* of all the literals are scaled down.

When the branching component returns a variable v, the LSIDS-based phase selection return positive if $activity[v] > activity[\neg v]$, and negative otherwise. One

```
void litBumpActivity(Lit l, double mult)
  activity[l] += inc ∗ mult
  if (activity[l] > 1e100)
    litRescore()

void litDecayActivity()
  inc ∗= 1/decay

void litRescore()
  for (int i = 0; i < nVars(); i++)
    activity[i] ∗= 1e−100
    activity[¬i] ∗= 1e−100
    inc ∗= 1e−100
```

Fig. 2: Bump, Decay and Rescore procedures for LSIDS activity.

can view the proposed scheme as an attempt to capture both the participation of literals in learnt clause generation, in spirit similar to VSIDS, and storing the information about trend, à la phase saving/decay polarity score.

5.1 Implementation Details

Figure 2 shows the methods to bump and decay the LSIDS scores. Figure 3 shows blocks of code from MiniSat, where the activity of literals is bumped. Figure 4 showcases the subroutine to pick the branching literal based on LSIDS. Of particular note is the setting of *mult* to 2 for assignment-based bumping while setting *mult* to 0.5 for Reason-based bumping. In order to maintain consistency with constants in EVSIDS, the constants in *litRescore* are the same as that of EVSIDS employed in the context of branching in mldc.

5.2 Experimental Results

To test the efficiency of LSIDS as a phase selection heuristic, we implemented[4] the heuristic on mldc, replacing the existing phase saving heuristic. We call the mldc augmented with LSIDS phase selection heuristic as mldc-lsids-phase. Similar to the previous section; we tested the implementations on SAT19 benchmarks using the setup mentioned in subsection 2.2.

Solved Instances and PAR-2 Score Comparison Table 3 compares numbers of instances solved by the solver mldc and mldc-lsids-phase. First, observe that mldc-lsids-phase solves 243 instances in comparison to 237 instances solved by mldc, which amounts to the improvement of 6 in the number of solved instances. On a closer inspection, we discovered that mldc performs CB for at least 1% of

[4] https://github.com/meelgroup/duriansat/tree/lsids

```
Bump literal scores for literals in learnt clause
void Solver.analyze(Constr confl)
  c : conflict clause
  litDecayActivity()
  for (int j = 0; j < c.size(); j++)
    Lit q = c[j];
    litBumpActivity(¬q, .5);

Bump literal scores when deleting assignment
void Solver.cancelUntil(int bLevel)                ▷ bLevel : backtrack level
  - for all elements which are getting cancelled
  for (int c = trail.size()−1; c >= trailLim[bLevel]; c−−)
    Var x = var(trail[c])
    Lit l = mkLit(x, polarity[x])
    litBumpActivity(¬l, 2);
```

Fig. 3: Sections in MiniSat like code, where LSIDS score is bumped and decayed.

```
Lit pickBranchLit()
    next : variable returned by branching heuristic
    CBT : denotes whether the last backtrack was chronological

  if (CBT)
      bool pol = pickLsidsBasedPhase(next)
      return mkLit(next, pol)
  else
      return mkLit(next, polarity[next])

bool pickLsidsBasedPhase( Var v)
  if ( activity[posL] > activity[negL] )
    return true
  else
    return false
```

Fig. 4: Method to choose branching literal

backtracks only in 103 instances out of 400 instances. Since mldc-lsids-phase is identical to mldc for the cases when the solver does not perform chronological backtracking, the improvement of 6 instances is out of the set of roughly 100 instances. It perhaps fits the often quoted paraphrase by Audemard and Simon [4]: *solving 10 or more instances on a fixed set (of size nearly 400) of instances from a competition by using a new technique, generally shows a critical feature.* In this context, we would like to believe that the ability of LSIDS-based phase selection

System	SAT	UNSAT	Total	PAR-2
mldc	140	97	237	4607.61
mldc-lsids-phase	147	96	243	4475.22

Table 3: Performance comparison of LSIDS based phase selection with phase saving on 400 SAT19 instances.

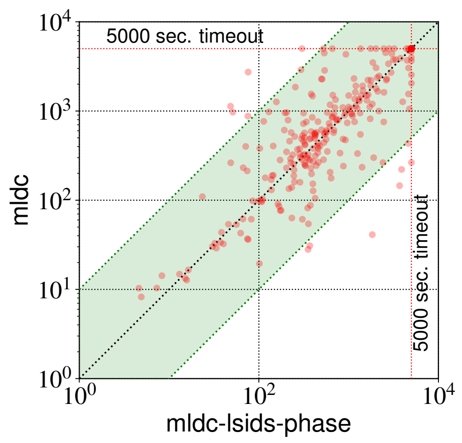

Fig. 5: Performance comparison of mldc-lsids-phase vis-a-vis mldc

to achieve improvement of 6 instances out of roughly 100 instances qualifies LSIDS-base phase saving to warrant serious attention by the community.

Table 3 also exhibits enhancement in PAR-2 score due to LSIDS-based phase selection. In particular, we observe mldc-lsids-phase achieved reduction 2.87% in PAR-2 score over mldc, which is significant as per SAT competitions standards. In particular, the difference among the winning solver and runner-up solver for the main track in 2019 and 2018 was 1.27% and 0.81%, respectively. In Figure 5, we show the scatter plot comparing instance-wise runtime performance of mldc vis-a-vis mldc-lsids-phase. While the plot shows that there are more instances for which mldc-lsids-phase achieves speedup over mldc than vice-versa, the plot also highlights the existence of several instances that time out due to the usage of mldc-lsids-phase but could be solved by mldc.

Solving Time Comparison. Figure 6 shows a cactus plot comparing performance of mldc and mldc-lsids-phase on SAT19 benchmarks. We present number of instances on x-axis and the time taken on y-axis. A point (x, y) in the plot denotes that a solver took less than or equal to y seconds to solve y benchmarks out of the 400 benchmarks in SAT19. The curves for mldc and mldc-lsids-phase indicate, for

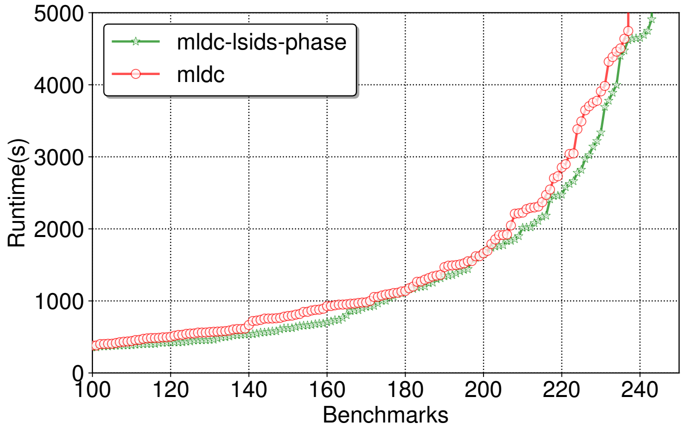

Fig. 6: Each of the curve corresponds to the performance of a solver, by means of number of instances solved within a specific time. At a specific runtime t, a curve to further right denotes the solver has solved more instances by time t.

.

every given timeout, the number of instances solved by mldc-lsids-phase is greater than or equal to mldc.

Percentage of usage and difference of selected phase. Among the instances solved by mldc-lsids-phase, percentage of decisions taken with LSIDS phase selections is on average 3.17% over the entire data set. Among the decisions taken with LSIDS phase selection, the average fraction of decisions where the selected phase differs from that of phase saving is 4.67%; It is worth remarking that maximum achieved is 88% while the minimum is 0%. Therefore, there are benchmarks where LSIDS and phase selection are entirely the same while there are benchmarks where they agree for only 12% of the cases. The numbers thereby demonstrate that the LSIDS-based phase selection can not be simply simulated by random or choosing phase opposite of phase selection.

Applicability of LSIDS in NCB-state. The performance improvements owing the usage of LSIDS during CB-state raise the question of whether LSIDS is beneficial in NCB-state as well. To this end, we augmented mldc-lsids-phase to employ LSIDS-based phase selection during both NCB-state as well as CB-state. Interestingly, the augmented mldc-lsids-phase solved 228 instances, nine less compared to mldc, thereby providing evidence in support of our choice of usage of LSIDS during CB-state only.

Deciding the best combination of CB and NCB. Nadel and Ryvchin [29] inferred that SAT solvers benefit from an appropriate combination of CB and NCB rather than solely reliance on CB or NCB. To this end, they varied two parameters, T and C according to the following rules to heuristically decide the best combination.

		T = 100				C = 4000			
		C = 2000	3000	4000	5000	T = 25	90	150	200
SAT	mldc	137	**141**	140	137	139	137	134	138
	mldc-lsids-phase	**143**	139	**147**	**139**	**142**	**141**	**139**	**142**
UNSAT	mldc	98	96	97	97	98	**97**	95	97
	mldc-lsids-phase	**99**	**101**	**96**	**100**	**99**	96	**99**	97
Total	mldc	235	237	237	234	237	233	229	235
	mldc-lsids-phase	**242**	**240**	**243**	**239**	**241**	**238**	**238**	**239**
PAR-2	mldc	4663	4588	4607	4674	4609	4706	4773	4641
	mldc-lsids-phase	**4506**	**4558**	**4475**	**4575**	**4555**	**4556**	**4622**	**4583**

Table 4: Performance comparison of LSIDS based phase selection with phase saving on 400 SAT19 instances with different T and C.

- If the difference between the current decision level and backtrack level returned by conflict analysis procedure is more than T, then perform CB.
- For the first C conflicts, perform NCB. This rule supersedes the above one.

Following the process, we experimented with different sets of T and C to determine the best combination of T and C for mldc-lsids-phase. For each configuration (T and C), we computed the performance of mldc too. The results are summerized in Table 4. It turns out that $T = 100, C = 4000$ performs best in mldc-lsids-phase. Interestingly, for most of the configurations , mldc-lsids-phase performed better than mldc.

5.3 Case Study on Cryptographic Benchmarks

Following SAT-community traditions, we have concentrated on SAT-19 benchmarks. But the complicated process of selection of benchmarks leads us to be cautious about confidence in runtime performance improvement achieved by LSIDS-based phase selection. Therefore, in a bid to further improve our confidence in the proposed heuristic, we performed a case study on benchmarks arising from security analysis of *SHA-1 cryptographic hash functions*, a class of benchmarks of special interest to our industrial collaborators and to the security community at large. For a message $\mathcal{M}$, a cryptographic hash function F creates a hash $\mathcal{H} = F(\mathcal{M})$. In a preimage attack, given a hash $\mathcal{H}$ of a message $\mathcal{M}$, we are interested to compute the original message $\mathcal{M}$. In the benchmark set generated, we considered SHA-1 with 80 rounds, 160 bits for hash are fixed, and k bits out of 512 message bits are fixed, $485 < k < 500$. The solution to the preimage attack problem is to give the remaining $(512 - k)$ bits. Therefore, the brute complexity of these problems will range from $O(2^{12})$ to $O(2^{27})$. The CNF encoding of these problems was created using the SAT instance generator for SHA-1 [30]. Note that by design, each of the instances is satisfiable. In our empirical evaluation, we

System	Total solved	PAR-2
mldc	291	9939.91
mldc-lsids-phase	299	9710.42

Table 5: Performance comparison of LSIDS based phase selection with phase saving on 512 cryptographic instances. Name of systems are same as Table 3.

focused on a suite comprising of 512 instances[5] and every experiment consisted of running a given solver with 3 hours of timeout on a particular instance.

Table 5 presents the runtime performance comparison of mldc vis-a-vis mldc-lsids-phase for our benchmark suite. First, we observe that mldc-lsids-phase solves 299 instances in comparison to 291 instances solved by mldc, demonstrating an increase of 8 instances due to LSIDS-based phase selection. Furthermore, we observe a decrease of 229 in PAR-2 score, corresponding to a relative improvement of 2.30%, which is in the same ballpark as the improvement in PAR-2 score observed in the context of SAT-19 instances.

6 Conclusion

In this paper, we evaluated the efficacy of phase saving in the context of the recently emerged usage of chronological backtracking in CDCL solving. Upon observing indistinguishability in the performance of phase saving vis-a-vis random polarity selection, we propose a new score: Literal State Independent Decay Sum (LSIDS) that seeks to capture both the activity of a literal arising during clause learning and also the history of polarities assigned to the variable. We observed that incorporating LSIDS to Maple_LCM_Dist_ChronoBTv3 leads to 6 more solved benchmarks while attaining a decrease of 132 seconds in PAR-2 score. The design of a new phase selection heuristic due to the presence of CB leads us to believe that the community needs to analyze the efficiency of heuristics for other components in the presence of chronological backtracking.

Acknowledgment We are grateful to the anonymous reviewers for constructive comments that significantly improved the final version of the paper. We are grateful to Mate Soos for identifying a critical error in the early drafts of the paper. This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme[NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore [1].

[5] Benchmarks are available at https://doi.org/10.5281/zenodo.3817476.

References

1. ASTAR, NTU, NUS, SUTD: National Supercomputing Centre (NSCC) Singapore (2018), `https://www.nscc.sg/about-nscc/overview/`
2. Audemard, G., Simon, L.: Glucose: a solver that predicts learnt clauses quality. SAT Competition pp. 7–8 (2009)
3. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Twenty-first International Joint Conference on Artificial Intelligence (2009)
4. Audemard, G., Simon, L.: Refining restarts strategies for sat and unsat. In: International Conference on Principles and Practice of Constraint Programming. pp. 118–126. Springer (2012)
5. Audemard, G., Simon, L.: On the glucose sat solver. International Journal on Artificial Intelligence Tools **27**(01), 1840001 (2018)
6. Biere, A.: Lingeling, plingeling and treengeling entering the sat competition 2013. Proceedings of SAT competition **2013**, 1
7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: International conference on tools and algorithms for the construction and analysis of systems. pp. 193–207. Springer (1999)
8. Biere, A., Fröhlich, A.: Evaluating cdcl variable scoring schemes. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 405–422. Springer (2015)
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176. Springer (2004)
10. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the third annual ACM symposium on Theory of computing. pp. 151–158 (1971)
11. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5**(7), 394–397 (1962)
12. Eén, N., Sörensson, N.: An extensible sat-solver. In: International conference on theory and applications of satisfiability testing. pp. 502–518. Springer (2003)
13. Frost, D., Dechter, R.: In search of the best constraint satisfaction search. In: AAAI. vol. 94, pp. 301–306 (1994)
14. Gomes, C.P., Selman, B., Kautz, H., et al.: Boosting combinatorial search through randomization. AAAI/IAAI **98**, 431–437 (1998)
15. Heule, M.J., Järvisalo, M., Suda, M.: Proceedings of sat race 2019: Solver and benchmark descriptions (2019)
16. Huang, J., et al.: The effect of restarts on the efficiency of clause learning. In: IJCAI. vol. 7, pp. 2318–2323 (2007)
17. Järvisalo, M., Le Berre, D., Roussel, O., Simon, L.: The international sat solver competitions. Ai Magazine **33**(1), 89–92 (2012)
18. Kautz, H.A., Selman, B., et al.: Planning as satisfiability. In: ECAI. vol. 92, pp. 359–363. Citeseer (1992)
19. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for sat solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 123–140. Springer (2016)
20. Liang, J.H., Ganesh, V., Zulkoski, E., Zaman, A., Czarnecki, K.: Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. In: Haifa Verification Conference. pp. 225–241. Springer (2015)
21. Liang, J.H., Oh, C., Mathew, M., Thomas, C., Li, C., Ganesh, V.: Machine learning-based restart policy for cdcl sat solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 94–110. Springer (2018)

22. Liang, J.H., Poupart, P., Czarnecki, K., Ganesh, V.: An empirical study of branching heuristics through the lens of global learning rate. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 119–135. Springer (2017)
23. Luo, M., Li, C.M., Xiao, F., Manya, F., Lü, Z.: An effective learnt clause minimization approach for cdcl sat solvers. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence. pp. 703–711 (2017)
24. Lynce, I., Marques-Silva, J.: Sat in bioinformatics: Making the case with haplotype inference. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 136–141. Springer (2006)
25. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning sat solvers. In: Handbook of satisfiability, pp. 131–153. ios Press (2009)
26. Marques-Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5), 506–521 (1999)
27. Möhle, S., Biere, A.: Combining conflict-driven clause learning and chronological backtracking for propositional model counting. EPiC Series in Computing **65**, 113–126 (2019)
28. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th annual Design Automation Conference. pp. 530–535. ACM (2001)
29. Nadel, A., Ryvchin, V.: Chronological backtracking. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 111–121. Springer (2018)
30. Nossum, V.: Instance generator for encoding preimage, second-preimage, and collision attacks on sha-1. Proceedings of the SAT competition pp. 119–120 (2013)
31. Oh, C.: Improving SAT solvers by exploiting empirical characteristics of CDCL. Ph.D. thesis, New York University (2016)
32. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: International conference on theory and applications of satisfiability testing. pp. 294–299. Springer (2007)
33. Ryvchin, V., Nadel, A.: Maple lcm dist chronobt: Featuring chronological backtracking. Proceedings of SAT competition p. 29 (2018)
34. Shtrichman, O.: Tuning sat checkers for bounded model checking. In: International Conference on Computer Aided Verification. pp. 480–494. Springer (2000)
35. Silva, J.P.M., Sakallah, K.A.: Graspa new search algorithm for satisfiability. In: The Best of ICCAD, pp. 73–89. Springer (2003)
36. Silva, J.M., Sakallah, K.A.: Conflict analysis in search algorithms for satisfiability. In: Proceedings Eighth IEEE International Conference on Tools with Artificial Intelligence. pp. 467–469. IEEE (1996)
37. Wetzler, N., Heule, M.J., Hunt, W.A.: Mechanical verification of sat refutations with extended resolution. In: International Conference on Interactive Theorem Proving. pp. 229–244. Springer (2013)