Empirical Insights Driven CDCL SAT Algorithms

by

Md Solimul Chowdhury

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

Boolean Satisfiability (SAT) is a well-known NP-complete problem. Despite the theoretical hardness of SAT, backtracking search based Conflict Directed Clause Learning (CDCL) SAT solvers can solve very large real-world SAT instances with surprising efficiency. The high efficiency of CDCL SAT solving is due to the careful integration of its key ingredients: preprocessing, inprocessing, decision heuristics, learning of clauses from conflicts, intelligent backtracking, and restarts.

Clause learning from conflicts helps a CDCL solver to prune its search space and achieve solving efficiency. Since finding conflicts is the only way to learn clauses, generating conflicts at a high rate is crucial for CDCL SAT solving.

A key component of CDCL SAT is the *decision step*, which heuristically selects an unassgined variable to make a boolean assignment during the search. Standard CDCL heuristics for branching are designed based on a look-back principle that uses information of past search states. Examples are Variable State Independent Decaying Sum (VSIDS) and Learning Rate Based (LRB). Both of these heuristics are conflict guided and prioritize selection of variables that are likely to lead to the discovery of conflicts.

These decision heuristics play a crucial role for CDCL. However, there is a lack of empirical understanding of how these branching heuristics work. This has been regarded as an important open problem in SAT research.

In this thesis, we study state-of-the-art CDCL decision heuristics empirically and design extensions of these heuristics based on the insights obtained from our empirical studies. First, we present a study on two types of decision variables: *glue* and *non-glue* variables. These are named depending on their appearance in a special type of learned clause called *glue-clause*. We demonstrate that decisions with glue variables are more conflict efficient than decisions with non-glue variables. Based on this insight, we develop a decision strategy named *glue bump*, which prioritizes selection of glue-variables. We show that the glue bumping strategy implemented on top of state-of-the-art CDCL SAT solvers improves the performance of these solvers. Secondly, we present a study of the conflict generation patterns in CDCL with two leading CDCL branching heuristics. We discovered that conflicts in CDCL are generated in phases of short bursts, often followed by longer conflict depression phases, where the search does not find any conflicts for a number of consecutive decisions. We developed a CDCL algorithmic extension named expSAT that performs random exploration during substantial conflict depression phases. We demonstrated that expSAT improves the performance of state-of-the-art CDCL SAT solvers on two years of SAT competition benchmarks and on a set of hard cryptographic instances from Bitcoin mining benchmarks. Thirdly, we study conflict producing decisions in CDCL, where we distinguish two types of decisions: single conflict (sc) and multi-conflict (mc) decisions. We present a characterization of sc and mc decisions, based on the quality of learned clauses that each produces. With this characterization, we propose a decision strategy named *Common Reason Variable score Reduction (CRVR)*. CRVR de-prioritizes selection of those variables, which contribute to the generation of lower quality learned clauses in *poor* mc decisions. Our empirical evaluation of CRVR demonstrates performance improvement of state-of-the-art CDCL SAT solvers on Satisfiable instances.

If there really were a machine with $\Phi(n) \approx k.n$ (or even $\approx k.n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine[1].

- Kurt Gödel

The SAT problem is at the core of arguably the most fundamental question in computer science: What makes a problem hard?

- Stephen Cook

The SAT problem is evidently a killer app, because it is key to the solution of so many other problems.

- Donald Knuth

[1]In the year of 1956, when he was in his deathbed, *Kurt Gödel*, one of the most brilliant minds in the history of Mathematical Logic, wrote a letter to *John von Neumann*, the icon of genius in many fields of intellectual inquiry including computer science. In that letter, Gödel hinted about the possibility of an efficient algorithm for solving the decidable fragment of the famous *Entscheidungsproblem* (i.e., the Decision Problem), which this quote embodies. We still do not know if there exists an easy solution to decision problems, in general (i.e., the question $\mathbf{P} \overset{?}{=} \mathbf{NP}$). The SAT problem is the most intensely studied decision problem and has central importance in resolving the above question.
A copy of that letter can be found in: `https://www.anilada.com/notes/godel-letter.pdf`

Dedication
I dedicate my PhD thesis to all of my teachers and mentors, including my mother *Azizunnessa Shenakta* and father *Md Mahmudul Chowdhury*, who first taught me how to read and write.

Acknowledgement

First and foremost, I would like to thank my supervisors *Jia-Huai You* and *Martin Müller* for their support and guidance throughout my PhD endeavour. I consider myself immensely lucky to get supervised by both of them, who possess extensive experience, knowledge, and wisdom in the area of heuristic search, logic and reasoning. During my PhD pursuit both have granted me abundance of freedom to pursue research projects, while providing me guidance when I was hitting dead ends. I believe that without their kind supervisions, I would not be able to complete this PhD thesis.

I would like to thank my thesis examiners *Randy Goebel*, *Lili Mou*, and *Peter van Beek* for examining my dissertation and giving me important feedback at my PhD defense.

During my candidacy exam, my examiners *James Wright* and *Omid Ardakanian* gave me valuable feedback that helped me to better formulate my research questions. I would like thank both of them for their feedback.

I would like to express my gratitude to *Ting-Han Wei*, a post doctoral fellow of Martin Müller, who gave me important suggestions on how to improve my oral presentation skills.

Time-to-time *Steve Sutphen* helped me with setting up the computational environments, where I performed my experiments. His efforts are appreciated.

Over the past few years, I have received constructive feedback from the reviewers of my conference/journal submissions, which helped me improve my PhD work. I would like to express my sincere gratitude to these *anonymous* peer reviewers.

I would like to take this opportunity to thank my Bachelor thesis supervisor *Sardar Anisul Haque*, for introducing me to the amazing world of SAT and his encouragement for pursuing graduate studies.

I am indebted to my wife *Sirazam Munira* for providing me strong mental support, especially during the darker times of my PhD study. I would also like to thank my 5 years old daughter *Umaiza Nereid Chowdhury*. Her immaculate face somehow gave me strengths to overcome hurdles of the rainy days of my PhD pursuit.

Finally, I would like to thank *Natural Science and Engineering Research Council (NSERC)*, *Alberta Innovates and Technology Features (AITF)* and *University of Alberta* for providing me with the financial support in the form of multiple scholarships.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Prologue

Given a formula $\mathcal{F}$ of boolean variables, the problem of Boolean Satisfiability (SAT) is to either determine variable assignments that satisfy $\mathcal{F}$, or report UNSAT in case no such assignment exists [11]. SAT is known to be NP-complete [19], which implies that solving SAT is considered intractable.

Complete SAT solvers based on the Davis-Putnam-Logemann-Loveland (DPLL) framework [20] employ a heuristics-guided strategy of backtracking tree search. At each node in the search tree, the DPLL algorithm selects a variable and searches two branches that correspond to the two boolean values of that variable. Solvers such as GRASP [72] and Chaff [49] substantially enhanced the DPLL framework by adding conflict analysis and clause learning. They are called *conflict directed clause learning* (CDCL) SAT solvers. Despite the theoretical hardness of SAT, modern CDCL SAT solvers have become the enabling technology for many real-world problems, such as hardware design verification [30], software debugging [15], classical planning [60] and encryption [48, 74]. This is the result of a careful combination of its key components, such as preprocessing [22, 34] and inprocessing [35, 46], robust branching heuristics [42, 43, 49], efficient restart policies [4, 55], intelligent conflict analysis [72], and effective clause learning [49].

The key decision-making step in a CDCL SAT solver is the selection of a variable from the current set of unassigned variables using a *decision heuristic*. Variable State Independent Decaying Sum (VSIDS) [49] and as well as its variants Learning rate Branching (LRB) [43] and Conflict History Branching (CHB) [42] are important state-of-the-art decision heuristics. A decision step is followed by a value assignment step, which assigns a boolean value to the decision variable using a *polarity selection heuristic*. Most state-of-the-art CDCL solvers use *phase-saving* as their predominant polarity selection heuristic [56]. It assigns the value to which a variable was last assigned.

Clause learning prunes the search space. Since discovery of conflicts is the only way to learn clauses, fast conflict discovery is critical for CDCL SAT solvers. The design of state-of-the-art CDCL decision heuristics conform to this goal. For example, the VSIDS, LRB and CHB heuristics reward and prioritize selection of variables that are involved in recent conflicts. The intuition is that selection of these variables is likely to generate further conflicts, leading to learned clauses that effectively prune the search space.

Conflict guided decision heuristics play a crucial role in allowing CDCL solvers to solve large real-world problems. As efficient as these decision heuristics are, understanding the inner-workings of these heuristics remains an open research issue. In the *Theoretical Foundations of Applied SAT Solving* workshop, the lack of empirical understanding of how CDCL decision heuristics work, was highlighted as an important open research problem in SAT [3] .

This thesis improves the understanding of the innerworkings of two leading CDCL decision heuristics `VSIDS` and `LRB` with respect to their conflict generation patterns. This led to the discovery of a series of important insights. For each insight, we develop algorithmic extensions of the standard CDCL algorithm. Some of these extensions have improved state-of-the-art SAT solvers on multiple years of SAT competition benchmarks.

In the following, we summarize the contributions of this thesis.

1.2 Contributions of this Thesis

Chapter 3: Characterization of Glue Variable and GB Method

A state-of-the-art criterion to measure the importance of a learned clause in CDCL SAT solving is called *literal block distance* (LBD), the number of distinct decision levels in the clause. The lower the LBD score of a learned clause, the better is its quality. Learned clauses with a LBD score of 2 are called *glue clause* and possess high pruning power. In this Chapter, we relate glue clauses to decision variables. We show experimentally that branching decisions with variables appearing in glue clauses, called *glue variables*, are more conflict efficient than those with nonglue variables. This observation motivates the development of a CDCL variable bumping scheme, which increases the heuristic score of a glue variable based on its appearance count in the glue clauses that are learned so far by the search. Empirical evaluation shows the effectiveness of the new method on the main track instances from SAT Competitions 2017 and 2018 with four different state-of-the-art CDCL SAT solvers. Finally, we show that the frequency of learned clauses that are glue clauses can be used as a reliable indicator of solving efficiency for instances for which the standard performance metrics fail to provide a consistent explanation.

Chapter 4: Conflict Depression Phases

In this Chapter, we analyze how conflicts are generated over the course of a CDCL SAT search. Our study of the `VSIDS` and `LRB` branching heuristics shows that they typically generate conflicts in short bursts, followed by what we call a *conflict depression* (CD) phase in which the search fails to generate any conflicts for a number of consecutive decisions. Our analysis shows a weak correlation between solving hardness and the average length of a CD phase. In this Chapter, we also show a correlation between backjumping length and average CD phase length.

Chapter 5: Search Guidance with Random Exploration amid CD Phases

The lack of conflict during CD phases indicates that the variables which are currently ranked highest by the branching heuristic fail to generate conflicts. We propose an exploration strategy called `expSAT`, which randomly samples variable selection sequences in order to learn an updated heuristic from the generated conflicts. The goal is to escape from conflict depressions expeditiously. The branching heuristic deployed in `expSAT` combines these updates with the standard activity scores of `VSIDS` and `LRB`. An extensive empirical evaluation with five state-of-the-art CDCL SAT solvers demonstrates good-to-strong performance gains with the `expSAT` approach for benchmark instances from SAT Competitions 2017 and 2018, and impressive gains over a set of hard bitcoin mining instances.

Chapter 6: Conflict Bursts Phases

In this Chapter, we study conflict-generating decisions in CDCL in detail. We investigate the impact of single conflict (sc) decisions, which generate only one conflict, and multi-conflict (mc) decisions which generate two or more. We empirically characterize these two types of decisions based on the quality of their learned clauses. We also show an important connection between consecutive clauses learned within the same mc decision, where one learned clause triggers the learning of the next one forming a chain of clauses. We formulate the notion of conflict proximity as a similarity measure and show that conflicts in mc decisions are more closely related than consecutive conflicts generated from separate sc decisions. We develop a new decision strategy named Common Reason Variable Reduction (CRVR) that reduces the selection priority of some variables from the learned clauses of mc decisions. Our empirical evaluation of CRVR implemented in three leading solvers demonstrates performance gains in satisfiable instances from the main track of SAT Competition-2020.

1.3 Other Related Contributions

During the course of this thesis, we have developed a new SAT benchmark which entails some query on life in a cellular automaton, where evolution of life follows certain rules. The description of the benchmark is available in Appendix A.

1.4 Publications and SAT Competition Results

1.4.1 Publications

This dissertation is based on the following first-authored peer-reviewed/under-preparation publications:

Chapter 3

- **Md Solimul Chowdhury**, Martin Müller, and Jia You: Exploiting Glue Clauses to Design Effective CDCL Branching Heuristics. In Proceedings of 25th International Conference on Principles and Practice of Constraint Programming (CP-2019): 126-143.

Chapter 4 and 5

- **Md Solimul Chowdhury**, Martin Müller, and Jia You: Guiding CDCL SAT Search via Random Exploration amid Conflict Depression. In Proceedings of 34th AAAI conference on artificial intelligence (AAAI-2020): 1428-1435.
- **Md Solimul Chowdhury**, Martin Müller, and Jia-Huai You: Preliminary Results on Exploration-Driven Satisfiability Solving. In Proceedings of 32nd AAAI Conference on Artificial Intelligence (AAAI-2018, Student Abstract): 8069-8070.

Chapter 6

- **Md Solimul Chowdhury**, Martin Müller, and Jia You. A Deep Dive into Conflict Generating Decisions. (`https://arxiv.org/abs/2105.04595`)

Links to the source codes related to these publications are available in Appendix B.

1.4.2 SAT Competition Results

1. In SAT Competition-2021,[1]

- (**Bronze Medal** in the SAT and the Main Track) Our solver kissat_gb implements GB method on top of kissat_sat (the winner of SAT Competition-2020). It won the **third place** (second runnerup) in the SAT track (out of 48 participants), the Main track (out of 48 participants), and the NoLimits track (out of 57 participants) of the competition.

- Our solver kissat_CRVR_gb combines CRVR and GB method in the kissat_sat. It took the 2nd place(**first runner up**) in the Nolimits track (out of 57 participants) of the competition.

- Our solver cms_expV_gbL combines the expSAT and the GB method on top of the solver CryptoMiniSAT. It was **ranked 5th** (out of 48 participants) in the Crypto Track of the competition.

2. In SAT Race-2019 [2], our solver expMaple_CM_GCBumpOnlyLRB, which combines the expSAT and the GB method was

- 2nd (**first runner-up**) in the UNSAT track of the competition (based on solving speed).

- and 3rd (**second runner-up**) in the SAT+UNSAT track of the competition (based on solution count).

1.5 Organization

The remainder of this dissertation is organized as follows: The next Chapter presents preliminary background material. In Chapter 3, we present a conflict efficiency based characterization of decisions with glue variables, a decision scheme named

[1]The results of the SAT competition-2021 are available at: `https://satcompetition.github.io/2021/slides/ISC2021-fixed.pdf`. The detailed results are available at: `https://satcompetition.github.io/2021/results.html`.

[2]The results of the SAT Race-2019 are available at : `http://sat-race-2019.ciirc.cvut.cz/downloads/satrace19slides.pdf`. The detailed results of the competition are available at: `http://sat-race-2019.ciirc.cvut.cz/index.php?cat=results`.

the GB method and an empirical evaluation of the GB method. Chapter 4 introduces and formalizes the novel concept of the conflict depression phase and presents a series of empirical results that reveals a series of important insights. In Chapter 5, we present `expSAT`, an algorithmic extension of the CDCL framework that guides CDCL search via random exploration amid conflict depressions. We present extensive experimental evaluations of `expSAT` and detailed performance analysis of our experiments. Chapter 6 presents the detailed study of conflict generating decisions in CDCL, the novel branching strategy CRVR, and its experimental evaluation. Chapter 7 presents our conclusions, discusses the impact of this thesis, and points toward potential future directions.

Chapter 2

Preliminaries

In this Chapter, we review preliminary materials for this thesis. Here, we present relevant concepts from SAT, CDCL SAT solving, Random Walks, and Statistics.

2.1 The Boolean Satisfiability (SAT) Problem

SAT Formula A Boolean *variable* v takes two values: *true* and *false*. A *literal* is either a variable (v) or its negation ($\neg v$). A *clause* is a disjunction of literals. A SAT formula is a conjunction of clauses. For example, $\mathcal{F}$ below is a SAT formula with 5 variables, 9 literals and 4 clauses.

$$\mathcal{F} = (x_1 \vee \neg x_5) \wedge (\neg x_2 \vee \neg x_1) \wedge (x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_1 \vee x_5)$$

The Task of SAT Solving Given a SAT formula $\mathcal{F}$, the SAT solving task is to either determine a variable assignment that satisfies $\mathcal{F}$ (i.e, a proof that $\mathcal{F}$ is satisfiable or SAT) or reports the unsatisfiability of $\mathcal{F}$ in case no such assignment exists (i.e, a proof that $\mathcal{F}$ is unsatisfiable or UNSAT). For example, given the SAT formula $\mathcal{F}$, a SAT solver may find the solution $x_1 = false,\ x_2 = true,\ x_3 = true,\ x_4 = true,\ x_5 = false$, which satisfies $\mathcal{F}$.

SAT is a decision problem and well-known to be NP-Complete [19]. In general, we do not have an efficient procedure to decide SAT.

2.2 SAT Solvers

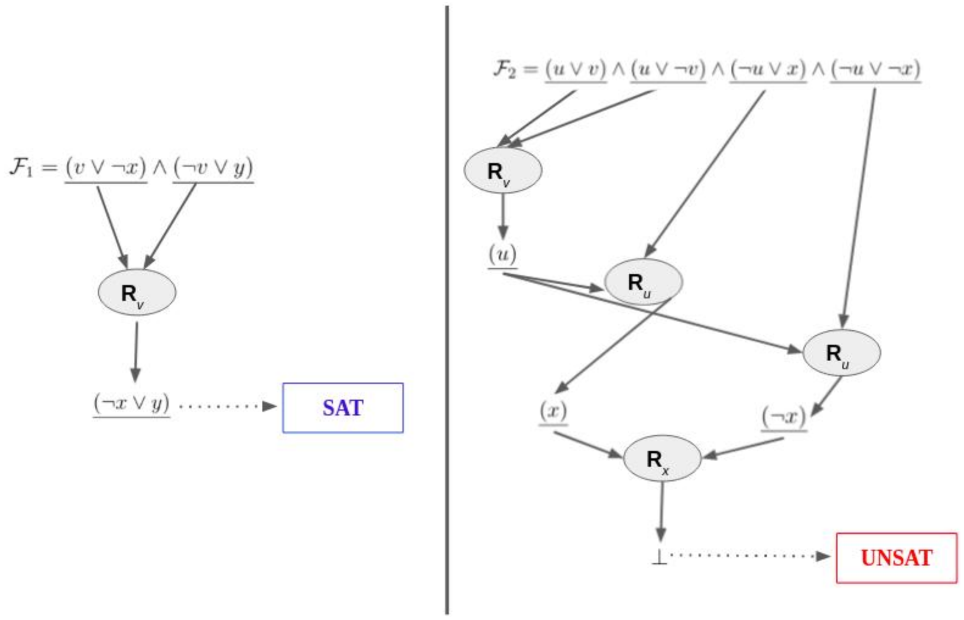
2.2.1 Davis-Putnam (DP) SAT Solvers

The first generation of SAT solvers were Davis-Putnam (DP) solvers [21]. DP employs an inference rule called *resolution* to decide a given SAT formula $\mathcal{F}$. Given two clauses $C = C_1 \cup a$ and $C' = C_2 \cup \neg a$, the resolution operation $\mathbf{R}_a$ applied on C and C' produces a new clause $C'' = C_1 \cup C_2$ by removing the clashing literals a and $\neg a$. The clause C'' is called the *resolvent* of $\mathbf{R}_a$, applied to C and C'.

DP applies resolution iteratively to eliminate variables of $\mathcal{F}$. Each iteration i produces formula $\mathcal{F}_i$ which is equisatisfiable to $\mathcal{F}$. At iteration i, if an empty clause is derived as a resolvent, then $\mathcal{F}$ is UNSAT, and $\mathcal{F}$ is SAT, if $\mathcal{F}_i$ does not contain an empty clause after performing all possible resolution steps.

The Figure 2.1 show examples of how a DP based SAT solver decides a SAT and UNSAT problem.

Figure 2.1: Examples of DP procedure. $\mathbf{R_a}$ denotes the resolution operation on variable a.


$\mathcal{F}_1 = (v \vee \neg x) \wedge (\neg v \vee y)$
R_v
$(\neg x \vee y)$
SAT
$\mathcal{F}_2 = (u \vee v) \wedge (u \vee \neg v) \wedge (\neg u \vee x) \wedge (\neg u \vee \neg x)$
R_v
(u)
R_u
R_u
(x)
$(\neg x)$
R_x
$\perp$
UNSAT

2.2.2 Davis–Putnam–Logemann–Loveland (DPLL) SAT Solvers

The resolution operation in DP may produce a large numbers of resolvents, causing an exponential blow up in the size of the formula in the worst case. A backtracking search based algorithm named DPLL [20] was proposed to solve this problem of exponential memory blow up. DPLL requires only a linear amount of memory of the size of the input formula in the worst case.

DPLL solvers build a heuristics-guided backtracking search tree to solve a given SAT formula by extending an initially empty *partial assignment*, a set of literals representing how the corresponding variables are assigned. In each *branching decision*, the solver extends the current partial assignment by selecting a variable v from the current set of unassigned variables, and assigns a boolean value p to v. v is called a *branching variable*. A branching decision is associated with a *decision level* ≥ 1, which denotes the depth of the search tree where the branching decision has taken place. After the assignment of v, *unit propagation* (UP) simplifies $\mathcal{F}$ by deducing a new set of implied variable assignments, which are added to the current partial assignment. After UP, the search moves down to the next decision level to make another branching decision.

UP may lead to a *conflict* due to a *falsified* or *conflicting* clause, which cannot be satisfied under the current partial assignment. In case of a conflict, DPLL backtracks upto the current decision level by undoing assignments made in the current decision level. It then assign $\neg p$ to v. After both values of v have been tried the search backtracks to the previous level and continues from there by assigning the complementary value to the decision variable at that level. Algorithm 1 shows a high-level pseudocode of DPLL, which is a simplified version of the DPLL procedure presented in [11].

2.2.3 CDCL SAT Solvers

In the late 1990s, much more powerful Conflict Driven Clause Learning (CDCL) SAT solvers such as GRASP [72] and Chaff [49] emerged. SAT solving in the CDCL framework is fundamentally inspired by the DPLL framework, but differs

Algorithm 1: Pseudocode of the DPLL Algorithm

```
Input: A CNF SAT formula: $\mathcal{F}$
Output: Satisfiability of $\mathcal{F}$
1  while unsolved do
2      if conflict then
3          Backtrack()
4      end
5      else if found_an_unit_clause then
6          UnitPropagate()
7      end
8      else
9          Decide()
10     end
11 end
```

substantially in the way it performs search. CDCL is still the state-of-the-art framework of SAT solving with structured real-world instances. CDCL alters the DPLL framework by adding sophisticated conflict-guided variable selection and clause learning, intelligent backtracking, and restarts.

Algorithm 2 shows the high-level pseudocode for the CDCL procedure. This procedure is a simplified version of the CDCL procedure presented in [11].

Once a conflict is reached by UP, CDCL performs *conflict analysis*. This step determines the root cause of a conflict and generates a *learned clause (lc)* that prevents the same conflicts from reappearing in the future, thereby pruning the search space. A *backjumping level (bl)* is computed from *lc*. If *bl* is 0 then the formula is UNSAT[1], otherwise, the search backtracks to *bl* and continues from there by making an assignment to a variable that appears in *lc*. At any given state, if all the clauses in $\mathcal{F}$ are satisfied by the assignments in $assign(\mathcal{F})$, then $\mathcal{F}$ is SAT with respect to $assign(\mathcal{F})$.

A CDCL SAT solver can learn clauses at a very fast rate. Keeping all the learned clauses in a clause database can quickly exhaust memory, and can lead to poor speed. Therefore, a solver routinely manages the clause database by deleting learned clauses that it considers unimportant.

[1]The decision made at the decision level *bl* is the root cause of the current conflict. If *bl* is 0, then it indicates that the current conflict is not caused by any decision and the formula is unsatisfiable.

Algorithm 2: Pseudocode of the CDCL SAT Algorithm

```
Input: A CNF SAT formula: $\mathcal{F}$
Output: Satisfiability of $\mathcal{F}$
while unsolved do
    if conflict then
        Analyze_Conflict()
        Learn()
        Backtrack()
    end
    else if found_an_unit_clause then
        UnitPropagate()
    end
    else if restart_needed then
        Restart()
    end
    else
        Decide()
    end
end
```

A CDCL SAT solver performs many *restarts* with an empty partial assignment. All other aspects of the current state of the search, such as the learned clauses, the heuristic scores and various parameter values are preserved at a restart. After a restart, the search starts building a new partial assignment.

The Process of Conflict Analysis and Clause Learning

Most state-of-the-art CDCL SAT solvers employ the *first Unique Implication Point (fUIP)* scheme to learn a clause [11]. Starting with conflicting clause C, fUIP continues to resolve literals from the current decision level until it finds a clause $L = R \vee \{\neg f\}$ such that the literal f was assigned in the current decision level, while all literals in R were assigned earlier. f is called the *fUIP literal* for the current conflict. The fuip literal f is contained in every path from the current decision variable to the current conflict. The literals in $\{r \mid \text{literal } r \in R\} \cup \{f\}$ are called *reason literals* for the current conflict, since their assignments caused the current conflict. We call R the *reason clause* for the current conflict with conflicting clause C.

L is learned and search backjumps to a *backjumping level* `bl` which is computed from L[2]. Before backtracking, the clause $L = R \vee \neg f$ is unsatisfied under the current partial assignment. After backtracking to `bl`, $\neg f$ is the only unassigned literal in L. The search proceeds by unit-propagating $\neg f$ from L. The assignment of $\neg f$ avoids the conflict at C, but may create further conflicts within the same decision. Please see [11] for a detailed overview of CDCL SAT Solving.

The Importance of Fast Conflict Generation

Whenever a conflict occurs during the search, a CDCL SAT solver learns a clause from that conflict. The learned clauses help to prune the search space, which has a huge impact on solving efficiency. In [57], Pipatsrisawat et al. showed that for UNSAT formulas, the shortest refutation proofs found by the CDCL SAT solving process with clause learning are only polynomially longer than shortest refutation proofs produced by *general resolution*, a powerful proof system.

In [41], Liang et al. showed that more efficient branching heuristics have the following empirical properties: on average, (a) they generate more conflicts per branching decision and (b) learned clauses from those conflicts are of higher quality. Therefore generating conflicts at a fast rate and learning high quality clauses from those conflicts are very important aspects of efficient CDCL SAT solving.

CDCL Branching Heuristics

The standard CDCL SAT branching heuristics are designed following the look-back principle. They prioritize the selection of variables which have been involved in recent conflicts. The intuition is that such variables will continue to generate more conflicts, if assigned again. Here, we briefly discuss VSIDS and LRB as representative CDCL branching heuristics.

VSIDS The *Variable State Independent Decaying Sum* (VSIDS), introduced by Moskewicz et al. in [49], is a popular family of dynamic branching heuristics.

[2]If `bl` is too far from the current decision level, then performing chronological backtracking results in better solving efficiency [50]. Most of the leading CDCL solvers employ a combination of chronological and non-chronological backtracking.

We focus on exponential VSIDS as a representative member, as presented in [49]. VSIDS maintains an *activity score* for each variable in a given SAT formula. During conflict analysis, VSIDS increases the activity score of each variable that is involved in conflict resolution, by a *variable bumping factor* g^z, where $g > 1$ is a constant and z is the count of the number of conflicts in the search so far. VSIDS puts a strong focus on variables that participated in the most recent conflicts.

LRB: In the *Learning Rate Branching* (LRB) branching heuristic [43], a variable v is regarded to become alive when it is assigned by a branching decision or propagation, and becomes dead when it is unassigned by backtracking. When v gets assigned (resp. unassigned), let z (resp. $z' > z$) be the count of number of conflicts in the search so far. z (resp. z') marks the birth (resp. death) of v. LRB tracks the participation count $P(v)$ of v in generation of learned clauses within the conflict interval $I(v) = z' - z$. When v is unassigned, LRB computes the *reward* $R(v) = \frac{P(v)}{I(v)}$, which is the rate of its participation in learned clause generation. An activity score for v is computed from $R(v)$. In search, the variable with maximum activity score is selected for branching.

CDCL Polarity Heuristics

Once a variable is decided by the decision heuristics, a CDCL SAT solver uses a *polarity heuristic* to assign a boolean value to the decided variable. The state-of-the-art polarity heuristic is the *phase-saving* heuristic, which we review below.

The Phase Saving Heuristic: Assume that v^+ and v^- are the two literals where variable v is assigned *true* and *false*, respectively. Also assume that at decision step i, a variable u is assigned, which creates a propagation with v^x, where $x \in \{+, -\}$ and that this propagation is followed by a conflict. After conflict analysis, during backtracking, the phase saving heuristic saves this last assigned polarity x of v in $polarity[v]$. At decision step $j > i$, assume that the CDCL decision heuristic selects v again. Then the phase saving heuristic selects the literal v^x [56].

In CDCL SAT solvers, such as Glucose [6] and Maple [44], $polarity[v]$ is initialized to $-$ for each variable v of a given SAT formula. This initialization to negative polarity is an artifact of the encoding of most SAT benchmarks, which generally produce formulas with more positive than negative literals [31].

Learning Rate, Learned Clause Quality and Glue Clauses

Here we review three relevant conflict metrics:

Global Learning Rate The Global Learning Rate (GLR) [41] is defined as $\frac{n_c}{n_d}$, where n_c is the number of conflicts generated in n_d decisions. GLR measures the average number of conflicts that a solver generates per decision.

The Literal Block Distance (LBD) Score The LBD score of a learned clause c [8] is the number of distinct decision levels in c. If LBD$(c) = n$, then c contains n propagation blocks, where each block has been propagated within the same branching decision. Intuitively, variables within a block are closely related, and learned clauses with lower LBD score tend to have higher quality [8, 41].

Glue Clauses Glue clauses [8] have a LBD score of 2. They are the most important types of learned clauses. A glue clause connects a literal from the current decision level with a block of literals assigned in a previous decision level. Glue clauses have more potential to reduce the search space more quickly than other learned clauses with higher LBD scores.

Glue to Learned (G2L) This measure represents the fraction of learned clauses that are glue clauses [18]. It is defined as $\frac{g}{c}$, if there are g glue clauses among c learned clauses.

State of the Art CDCL Solver Systems In recent years, three families of solvers dominated the SAT competitions.

- **The Glucose Family:** *Glucose* [6] and its numerous extensions use VSIDS and variants of VSIDS as their branching heuristic. Solvers from this family also use rapid restart strategies and maintain aggressive clause database cleaning schedules.

- **The Maple Family:** MapleCOMSPS [44] and its numerous variants use a hybridization of various branching heuristics, such as VSIDS, LRB and Dist [80]. Solvers from this family use less aggressive clause database reduction and restart policies than the *Glucose* family.

- **The CaDiCaL and Kissat Family:** CaDiCaL [12] and its descendant Kissat [13] are another family of high performance CDCL SAT solvers. In both solvers search and inprocessing are interleaved. Inprocessing is a powerful technique that simplifies the current clause collection. Both solvers use VSIDS and Variable Move To Front (VMTF) [62] as their branching heuristics. Both solvers perform ultra-rapid restarts during the search.

2.3 Exploration Methods

Nakhost et al. [51] proposed *Monte Carlo Random Walk* (MRW) in the context of deterministic planning. MRW determines the best possible action from a set of available actions by employing random walks in the local neighborhood of the current search state. The random exploration in `expSAT` presented in Chapter 5 is inspired by MRW. We review this algorithm below:

Monte Carlo Random Walk At a given state s of the search, MRW performs a fixed number of random walks in the local neighborhood of s. A walk consists of a fixed number of steps. The goal of exploration in MRW is to find a state s^*, with best heuristic score among the explored states. Search selects the sequence of actions that leads to s^* and repeats the process from there.

2.4 Statistical Correlationships

Here we review some elementary concepts from statistics, which are relevant for this thesis. For detailed overview of these concepts see [36].

Mean and Median

Given a population sample $X = (X_1, \ldots, X_n)$ of size n, the *sample mean* μ_X or *expected value* $\mathbf{E}[X]$ is defined as the arithmetic average of the population.

$$\mathbf{E}[X] = \mu_X = \frac{X_1 + \cdots + X_n}{n}$$

The *median* of a population sample is a number that is exceeded by at most half of the numbers and is preceded by at most half of the numbers in that population sample.

Variance, Standard Deviation and Covariance

The *variance* $\mathbf{VAR}(X)$ of a population sample is defined as the expected squared deviation from the mean.

$$\mathbf{VAR}(X) = \mathbf{E}[X - \mu_x]^2$$

The *standard deviation* σ_X of a population sample is the square root of its variance.

$$\sigma_X = \sqrt{\mathbf{VAR}(X)}$$

Pearson Correlation Coefficient

Given two population samples X and Y, the *covariance* $\mathbf{COV}(X, Y)$ between X and Y is the expected product of deviations of X and Y from their respective mean values.

$$\mathbf{COV}(X, Y) = \mathbf{E}[(X - \mu_X)(Y - \mu_Y)]$$

The *bi-variate correlation coefficient*, also called the *Pearson Correlation Coefficient* $\rho_{X,Y}$ between two population samples X and Y is defined as

$$\rho_{X,Y} = \frac{\mathbf{COV}(X, Y)}{\sigma_X \sigma_Y}$$

$\rho_{X,Y}$ measures the statistical association between two samples X and Y.

2.5 SAT Solvers, Hardware, and Test Sets

We used the following SAT solvers, test sets and hardware environments to conduct the experiments for this thesis:

- Chapter 3:
  - **SAT Solvers:** We used 4 state-of-the-art CDCL SAT solvers as baseline: glucose 4.1 [6], MapleCOMSPS_Pure_LRB [44], Maple_LCM_Dist [64], and MapleLCMDistChronoBT [65].
  - **Test Sets:** We used a single test set in this Chapter.
    - **Test Set 1** ($\mathbf{TS1}$) contains a total of 750 SAT instances: 350 instances from the main track of SAT Competition 2017 [63] and 400 instances from SAT-2018 [66]. We used a time limit of 5,000 seconds per instance.
  - **Hardware:** All experiments reported in this Chapter were run on a Linux workstation with 64 Gigabytes RAM and processor clock speed of 2.40 GHZ.
- Chapters 4 and 5:
  - **SAT Solvers:** In Chapter 4, we used the CDCL solvers glucose 4.2.1 [65] and MapleCOMSPS_Pure_LRB [44]. In Chapter 5, we added three more solvers: MapleCOMSPS [44] (winner of SAT Competition 2016), MapleCM [65] (3rd in SAT Competition 2018) and MapleLCMDistChronoBT [65] (winner of SAT Competition 2018).
  - **Test Sets:** In both Chapter 4 and 5, we used $\mathbf{TS1}$ as the primary test set. Unless specified otherwise, we used a timeout of 5,000 seconds for $\mathbf{TS1}$.

    In Chapter 5, we also used the following test set:
    - **Test Set 2** ($\mathbf{TS2}$) consists of 52 hard instances from the SATCoin (bitcoin mining) cryptographic benchmark, which are generated

with the instance generator from [47]. We generated these instances by varying the *range* parameter, which determines the difficulty of a SATCoin instance. For experiments with **TS2**, we set the time limit to 36,000 seconds per instance [3].

  - **Hardware:** All experiments in this Chapter were run on a Linux workstation with 48GB RAM and a processor clock speed of 2.93 GHz.

- Chapter 6:
  - **SAT Solvers:** We used 3 state-of-the-art CDCL SAT solvers in this Chapter: MapleLCMDiscChronoBT-DL-v3 [69] (the winner of SAT Race-2019), and the top two solvers in the main track of SAT Competition-2020, Kissat-sc2020-sat and Kissat-sc2020-default [67].
  - **Test Set:** The test set **TS3** consists of 400 benchmark instances from the main track of SAT Competition-2020 [67]. The timeout is 5,000 seconds per instance.
  - **Hardware:** Experiments were run on a Linux workstation with 64 Gigabytes of RAM and a processor clock speed of 2.4GHZ.

2.6 Solver Evaluation Criteria

For evaluating the baseline solvers and their extensions, we used the following two metrics:

Solution Count

The number of solved instances that a given solver solves from a fixed test set.

Penalized Average Runtime (PAR2) Score

A metric used in SAT competitions, the PAR2 score is defined as the sum of all runtimes for solved instances + $2 * timeout$ for unsolved instances; lower is better.

[3]**TS2** instances are available in `https://figshare.com/articles/TS2Instances/12579143`.

Chapter 3

Characterization of Glue Variables

3.1 Introduction

In CDCL a learned clause helps a solver to avoid search in a space, which does not contain any solution. As finding conflicts is the only way to learn a clause, fast conflict discovery is pivotal to the efficiency of SAT solving with CDCL. However, a large amount of learned clauses reduces the overall performance. The management of the learned clause database is also a key component of a modern CDCL SAT solver [49, 72]. A CDCL SAT solver routinely reduces the learned clause database during search by deleting learned clauses that it considers irrelevant.

In earlier CDCL SAT solvers, the size and recent activities of learned clauses were the dominant criteria for determining the relevance of learned clauses [23]. Glucose [8] was the first to apply a new measure called *literal block distance* (LBD), which indicates the number of distinct decision levels in a learned clause. *glue clauses* with a LBD score of 2, are of particular interest [8, 55] because they connect a block of closely related variables, and thus a relatively small number of decisions are needed to create a *unit clause* (i.e., a clause that has all but one literal assigned under the current partial assignment). Since a glue clause has potential to create unit clauses faster, it also has potential to generate conflicts faster with fewer numbers of decisions, which leads to pruning of the search space. For this reason, all modern CDCL SAT solvers permanently store glue clauses.

Inspired by these advantages of glue clauses, we study whether glue clauses can be used to help re-rank decision variables to improve search efficiency. We call the decision variables that have appeared in at least one glue clause up to the current search state *glue variables*, and all other variables *nonglue variables*.

The main contributions of this chapter are:

- We conduct an experiment using the 750 instances from the test set **TS1** (instances from SAT competition 2017 and 2018) with four state-of-the-art CDCL SAT solvers: glucose 4.1 (Glucose) [6], MAPLECOMSPS_PURE_LRB (MapleLRB) [44], Maple_LCM_Dist (MLD) [64] and MapleLCMDistChronoBT (MLD_CBT) [7]. Our experiment shows that decisions with glue variables are more conflict efficient than those with nonglue variables. Furthermore, glue variables are picked up by CDCL branching heuristics disproportionately more often.

- We design a variable score bumping method called *Glue Bumping* (GB), which dynamically bumps the activity score of a glue variable based on its current activity score and (normalized) *glue level*. The glue level counts the glue clauses in which the variable appears.

- We implemented and evaluated the GB method on top of the four SAT solvers mentioned above. For the 750 instances from **TS1**, all GB extensions solve more instances than their baseline and achieve lower PAR-2 scores. One of our extended solver solves 9 additional instances over the instances from SAT Competition 2017. According to [4], this level of performance gain closely resembles the introduction of a *critical feature*, which is remarkable, given the simplicity of the new method.

- We provide evidence that the frequency of glue clauses in learned clauses is a reliable indicator of solving efficiency. In [41], the authors reported correlations between solving efficiency of branching heuristics and standard metrics based on the global learning rate (GLR) and average LBD (aLBD) scores. Higher solving efficiency is indicated by higher average GLR and lower average aLBD. We show that these two measures do not provide a consistent explanation of solving efficiency for some subsets of **TS1**. However, a new measure G2L based on the fraction of learned clauses that are glue, we are able to provide a consistent explanation.

3.2 Notations

Let $\mathcal{F}$ be a SAT formula. Suppose a CDCL solver Ψ is solving $\mathcal{F}$ and is in current search state s. At s, Ψ has already taken $d > 0$ decisions and has learned a set of glue clauses. A *glue decision* is the branching decision that selects a glue variable and a *nonglue decision* is the branching decision that selects a nonglue variable. Suppose that until s, Ψ has taken gd glue decisions (resp. ngd nonglue decisions) which generated gc conflicts (resp. ngc conflicts).

- *Learning Rate* (LR): In contrast to the global learning rate GLR, the rate of conflict generation is over all decisions, we define learning rates over glue decisions only or over nonglue decisions only. The *LR with glue decisions* is defined as $\frac{gc}{gd}$, while *LR with nonglue decisions* is defined as $\frac{ngc}{ngd}$.
- *Average LBD* (aLBD): We define the average LBD score per conflict generated solely by glue decisions, or solely by nonglue decisions. Let $sumLBD_{gc}$ (resp. $sumLBD_{ngc}$) be the sum of LBD scores of the learned clauses derived from those gc (resp. ngc) conflicts. The *aLBD with glue decisions* (resp. *nonglue decisions)* is defined as $\frac{sumLBD_{gc}}{gc}$ (resp. $\frac{sumLBD_{ngc}}{ngc}$).

3.3 Conflict Efficiency of Glue Variables

In this section, we report an experiment that studies the role played by glue variables in CDCL SAT solving. We show that glue decisions are more conflict efficient (i.e., typically achieve higher average LR and lower average aLBD) than nonglue decisions. Additionally, we show that the branching heuristics of modern CDCL SAT solvers exhibit bias towards the selection of glue variables over nonglue variables.

The solvers in this experiment are Glucose, MapleLRB, MLD, and MLD_CBT. The branching heuristics used in the first two solvers are, respectively, VSIDS [49] and LRB [43]. In MLD, and MLD_CBT, the branching heuristics are based on a combination of three heuristics, VSIDS, LRB, and Dist [80], each of which is activated at different state of the search in these two solvers.

We run all 750 instances from **TS1** with 5,000 seconds timeout per instance. We instrumented the four solvers to collect the following statistics for each instance: (i) the number of glue and nonglue decisions, (ii) LR and aLBD for both glue and nonglue decisions, and (iii) the number of glue and nonglue variables. For each instance, all the measurements are taken at the final search state (i.e., either after satisfiability/unsatisfiability is determined or after timeout).

3.3.1 Conflict Generation Power of Glue Variables

Table 3.1 shows a comparison of average LR and average aLBD for glue and nonglue decisions, grouped by satisfiable, unsatisfiable and unsolved instances. Comparing columns D1 and D2, all solvers achieve significantly higher average LR with glue decisions. Columns E1 and E2 show that for all three categories of instances, MLD and MLD_CBT achieve significantly lower average LBD for glue decisions. For Glucose and MapleLRB, these values are roughly the same.

Table 3.1: Comparison of average LR (higher is better) and average aLBD (lower is better) for glue and nonglue decisions.

(A) Systems	**(B) Type**	**(C) #Inst**	**(D) Average of Learning Rate (LR)**		**(E) Average of aLBD**	
			(D1) Glue Decisions	**(D2) Nonglue Decisions**	**(E1) Glue Decisions**	**(E2) Nonglue Decisions**
Glucose	SAT	180	**0.55**	0.41	18.44	**18.18**
	UNSAT	191	**0.56**	0.44	**11.2**	11.4
	Unsolved	379	**0.57**	0.48	**24.76**	25.48
MapleLRB	SAT	194	**0.47**	0.38	20.18	**19.25**
	UNSAT	190	**0.58**	0.46	**11.92**	12.39
	Unsolved	366	**0.48**	0.44	34.86	**33.39**
MLD	SAT	235	**0.47**	0.19	**31.76**	40.55
	UNSAT	207	**0.59**	0.27	**12.8**	30.1
	Unsolved	308	**0.52**	0.37	**24.23**	34.09
MLD_CBT	SAT	238	**0.51**	0.21	**32.1**	41.9
	UNSAT	215	**0.61**	0.27	**13.17**	24.74
	Unsolved	297	**0.53**	0.37	**25.25**	36.7

Figure 3.1 plots the actual distribution of LR values for the 750 instances for all four solvers. For all solvers and for a large majority of the instances, glue decisions achieve higher LR than nonglue decisions.

Figure 3.2 shows per instance aLBD scores in log scale for the 750 instances for glue and nonglue decisions.

Figure 3.1: Comparison of LR values for glue and nonglue decisions. Instances are sorted by the LR values of glue decisions. The number at the top of each plot represents the percentage of instances, for which LR of glue decisions are higher than LR of nonglue decisions.

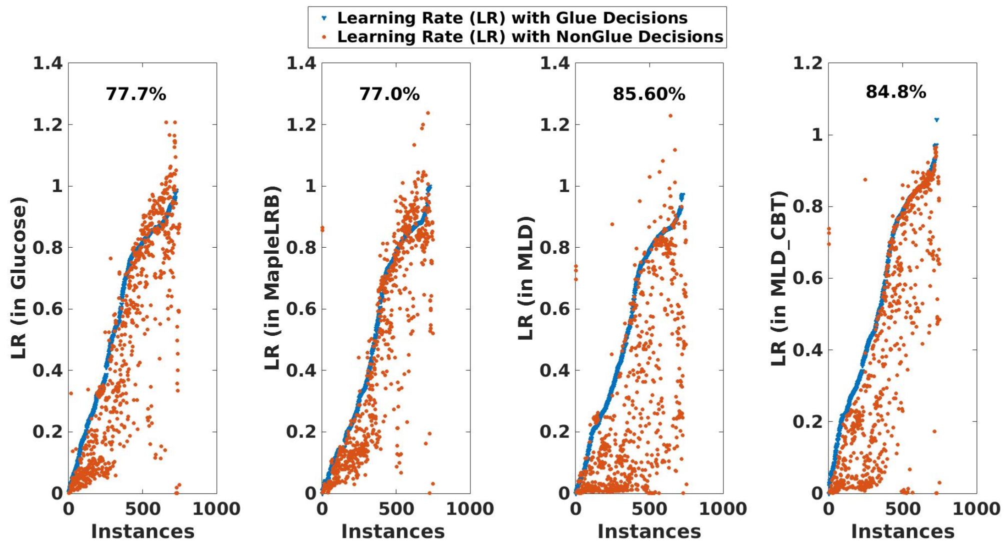
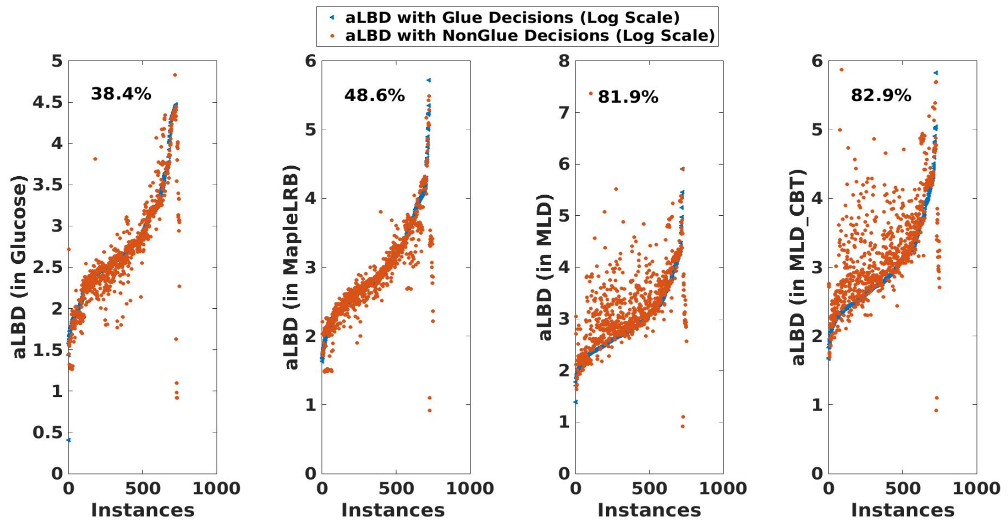


Figure 3.2: Comparison aLBD scores (in Log Scale). Instances are sorted by the aLBD of glue decisions. The number at the top of each plot represents the percentage of instances, for which aLBD of glue decisions are lower than aLBD of nonglue decisions.



- For Glucose and MapleLRB (first and second plots in Figure 3.2), for more than half of the instances, the aLBD score of the learned clauses by nonglue decisions is lower than the aLBD score of the learned clauses by glue decisions. The average values of aLBD under columns E1 and E2 in Table 3.1 for Glucose and MapleLRB reflect these ground data.

- MLD and MLD_CBT (third and fourth plots in Figure 3.2) show quite a different behavior. The aLBD scores of the learned clauses by glue decisions are lower for the large majority of instances. Again, the average values of aLBD in columns E1 and E2 in Table 3.1 for MLD and MLD_CBT reflect this ground data.

Overall, glue decisions are more conflict efficient than nonglue decisions for all the tested solvers. For the average aLBD metric, glue decisions in the winners of the last two SAT competitions, MLD and MLD_CBT, generate substantially lower (better) values.

3.3.2 Selection Bias towards Glue Variables in CDCL

We are interested in the question: Do conflict guided CDCL branching heuristics exhibit any bias towards glue variables over nonglue variables?

Given a SAT formula $\mathcal{F}$ and a solver Ψ, we define the *glue fraction* (GF) (resp. *nonglue fraction* (NF)) as the fraction of variables in $\mathcal{F}$ that are glue (resp. nonglue) variables, after Ψ completes its run with $\mathcal{F}$. GF (resp. NF) measures the pool size of glue (resp. nonglue) variables in $\mathcal{F}$ as a fraction of the total number of variables in $\mathcal{F}$.

Table 3.2 shows results over the 750 instances of $\mathbf{TS1}$. Column B lists the average GF and average percentage of glue decisions, while Column C shows the average nonglue fraction and the average percentage of nonglue decisions. For all four solvers, on average, the number of glue variables is significantly smaller than the number of nonglue variables (columns B1 and C1). For all the four solvers, on average, glue decisions relative to glue variables pool size are higher (Column B2) than nonglue decisions (Column C2) relative to the nonglue variables pool size.

Table 3.2: Biased Selection of Glue Variables

(A) Systems	(B) Average for Glue Variable		(C) Average for Nonglue Variables	
	GF (B1)	Glue Decisions % (B2)	NF (C1)	Noglue Decisions % (C2)
Glucose	0.25	65.43%	0.75	34.57%
MapleLRB	0.21	63.14%	0.69	36.86%
MLD	0.22	47.60%	0.78	52.60%
MLD_CBT	0.22	48.76%	0.78	51.24%

In summary, the four state-of-the-art CDCL SAT solvers make a much larger percentage of glue decisions against relatively smaller pools of glue variables. This shows the bias of these solvers towards selecting glue variables in branching decisions.

3.4 Activity Score Bumping for Glue Variables

From the above analysis, it is clear that decisions with glue variables generate conflict at faster rate than decisions with nonglue variables. How can we exploit this empirical characteristic for more efficient SAT solving? We present a score bumping method, called *Glue Bump* (GB), which bumps the activity score of glue variables. The amount of bumping for a glue variable depends on the appearance count of that variable in glue clauses and its current activity score.

Definition 1: ***(Glue Level)*** *Let G be the set of learned glue clauses until search state s. The glue level $gl(v)$ of a glue variable v, is the number of glue clauses in G in which v appears.*[1]

A higher glue level indicates higher potential to create conflicts.

3.4.1 The GB Method

By using the current activity scores and (normalized) glue levels of glue variables (we will comment on normalization shortly), the GB method bumps the activity

[1] We omit the parameter s since the glue level of a variable is always computed w.r.t. a underlying search state by default, without confusion.

scores of glue variables. This gives higher preference to recently active glue variables with high glue levels. The GB method is simple to implement and conveniently integrates with activity based standard CDCL heuristics.

The GB method modifies a baseline CDCL SAT solver Ψ, creating its GB extension Ψ^{gb}, by adding the following two procedures, which are called at different states of the search.

Alg. 1: Increase Glue Level

```
Input: A newly learned glue clause $\theta$
1   Foreach variable $v$ in $\theta$
2       $gl(v) \leftarrow gl(v) + 1$
3   End
```

Alg. 2: Bump Glue Variable

```
Input: $A\ glue\ variable\ \ v$
1  $b \leftarrow activity(v) * \left(\frac{gl(v)}{|G|}\right)$
2  $activity(v) \leftarrow activity(v) + b$
```

Increase Glue Level

Whenever Ψ^{gb} learns a new glue clause θ, the glue level of variable v in θ is increased by 1 (Alg. 1 , line 2).

Bump Glue Variable

Alg. 2 bumps a glue variable v. It computes the *bumping factor* b for v (line 1), from the current activity score and the normalized glue level (is explained below) of v. The bumping is performed by adding the bumping factor of v to the activity score of v, which becomes the new activity score for v (line 2).

Glue Level Normalization

The glue level of a glue variable can grow unboundedly with the discovery of more and more glue clauses. The activity score of a glue variable also grows, but at a different rate. Thus scaling the glue level is necessary.

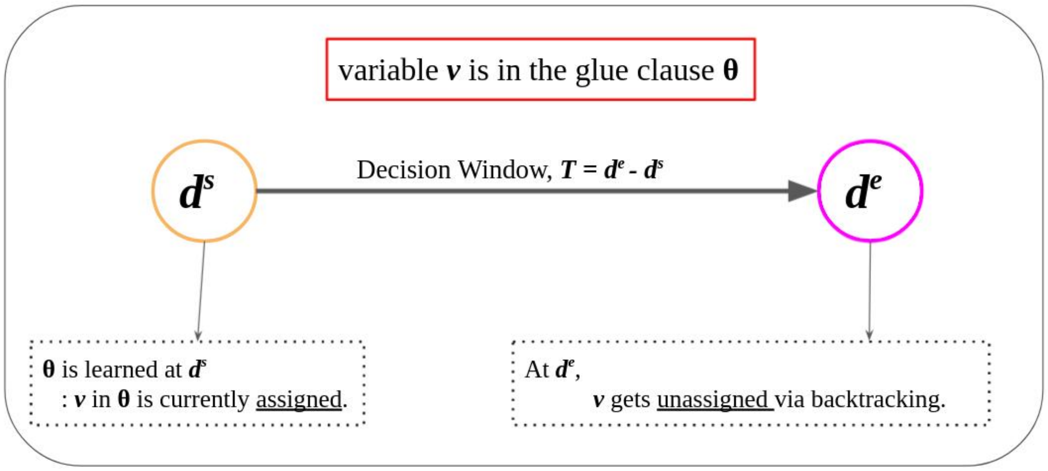
At a given state of the search, a given glue variable v appears in at least one glue clause. So, range of glue level of v is: $0 < gl(v) \leq |G|$. Hence, the normalized glue level $\frac{gl(v)}{|G|}$ is in the range (0,1].

The normalization scales the glue levels of glue variables by the *total number* of glue clauses discovered by the search so far.

Delayed Bumping of Glue Variables

Ψ^{gb} does not perform the bumping of v right after its hosting clause θ is discovered. It delays the bumping, until v is unassigned by backtracking. This is a subtle point which we explain below.

Figure 3.3: Delayed Bumping of Glue Variables. Let v be a glue variable that appears in the glue clause θ, which is learned at decision d^s. At d^s, all variables in θ including v are assigned. At decision $d^e > d^s$, assume that v gets unassigned via backtracking. Within the decision window $T = d^e - d^s$, possibly, (i) more glue clauses are learned. (ii) v gets involved in more conflicts. For these two events, b_{d^s}, the bumping factor computed at d^s will be different from b_{d^e}, where b_{d^e} is the more recent bumping factor for v.


variable **v** is in the glue clause **θ**
d^s
Decision Window, $T = d^e - d^s$
d^e
θ is learned at d^s
: v in θ is currently assigned.
At d^e,
v gets unassigned via backtracking.

- Let glue clause θ be the latest learned clause, such that all the variables in θ including v are assigned at the current search state. Any score bumping that v receives at this stage is not used until v gets unassigned.

- Let $T = d^e - d^s > 0$ be the decision window starting from the decision d^s that generates θ and ending at the decision d^e in which v gets unassigned. Within T, the search may generate more glue clauses which may contain v as well. v may also be involved in several conflicts during T, which increases its activity score. b_{d^e}, the bumping factor of v computed at d^e reflects a more recent measure than b_{d^s}, the one which is computed at d^s. By delaying the bumping of v until the decision d^e, when v is just unassigned and become a candidate variable for branching, the GB method boosts the activity score of v by a more recent bumping factor.

Figure 3.3 illustrates the difference of impact on computing the bumping factor for Glue variables with immediate and delayed bumping.

3.5 Implementation of GB and Experiments

3.5.1 Implementation of GB

We implemented the GB method on top of the CDCL SAT solvers Glucose, Maple LRB, MLD, and MLD_CBT and call the extended solvers Glucose^{gb}, MapleLRB^{gb}, MLD^{gb}, and MLD_CBT^{gb}, respectively. The baseline solvers do not distinguish between glue and nonglue variables, except Glucose, which bumps the activity scores of variables that are propagated from a glue clause.

In Glucose^{gb} and MapleLRB^{gb}, on the unassignment of a glue variable, the GB method updates the activity score of that glue variable in VSIDS and LRB, respectively. These are the heuristics used in their baselines. As remarked earlier, the baseline solvers MLD and MLD_CBT employ three heuristics, DIST, VSIDS and LRB, which are activated at different phases of the search. In any given phase, on the unassignment of a glue variable, MLD^{gb} and MLD_CBT^{gb} update the activity score of that glue variable for the currently active heuristic.

3.5.2 Experiments

We conduct experiments with our four extended solvers with $\mathbf{TS1}$ instances. We compare the extended solvers and their baselines in terms of number of solved instances, solution time and PAR-2 scores.

Solved Instances Comparison

Table 3.3 compares the four extended solvers with their baselines. Both MapleLRB^{gb} and MLD^{gb} solve 13 more instances (9 SAT, 4 UNSAT for the former and 11 SAT,

Table 3.3: Comparison of the four baseline solvers with their GB extensions for the instances from $\mathbf{TS1}$. The PAR-2 scores are scaled by 10^{-4}.

Systems	**SAT Comp-17**				**SAT Comp-18**				**SAT Comp-2017 and 2018**			
	SAT	**UNSAT**	**Total**	**PAR-2**	**SAT**	**UNSAT**	**Total**	**PAR-2**	**SAT**	**UNSAT**	**Total**	**PAR-2**
Glucose	83	96	179	1893	97	95	192	2274	180	191	371	4167
Glucose^{gb}	**86 (+3)**	96 (+0)	**182 (+3)**	**1868**	96 (-1)	**97 (+2)**	193 (+0)	**2273**	**182 (+2)**	**193 (+2)**	**375 (+4)**	**4141**
MapleLRB	80	95	175	1897	114	95	209	2069	194	190	384	3966
MapleLRB^{gb}	**87 (+7)**	**97 (+2)**	**184 (+9)**	**1824**	**117 (+3)**	**96 (+1)**	**213 (+4)**	**2027**	**204 (+10)**	**193 (+3)**	**397 (+13)**	**3851**
MLD	99	106	205	1635	136	101	237	1807	235	207	442	3442
MLD^{gb}	**103 (+4)**	**107 (+1)**	**210 (+5)**	**1593**	**143 (+7)**	**102 (+1)**	**245 (+8)**	**1725**	**246 (+11)**	**209 (+2)**	**455 (+13)**	**3318**
MLD_CBT	**103**	113	216	1565	135	**102**	237	1800	238	215	453	3365
MLD_CBT^{gb}	102 (-1)	**114 (+1)**	216 (+0)	**1539**	**138 (+3)**	101 (-1)	**239 (+2)**	**1756**	**240 (+2)**	215 (+0)	**455 (+2)**	**3295**

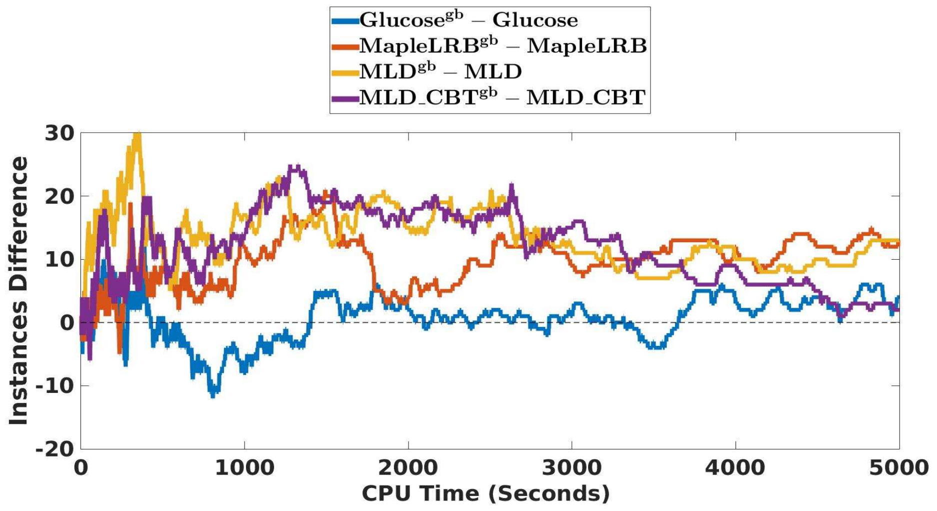
2 UNSAT for the latter). Glucose^{gb} solves 4 more instances (2 SAT, 2 UNSAT), and MLD_CBT^{gb} solves 2 additional instances (both SAT).

According to Audemard and Simon [4], solving 10 or more instances on a fixed set of instances from a competition by using a new technique generally shows a critical feature. MapleLRB^{gb} solves 9 more instances over the instances from SAT Competition 2017 and and MLD^{gb} solves 8 additional instances over the instances from SAT Competition 2018. The gains with MapleLRB^{gb} and MLD^{gb} are significant and come close to the introduction of a critical feature.

Solution Time Comparison

Figure 3.4 compares the performance of Glucose^{gb} (blue line), MapleLRB^{gb} (red line), MLD^{gb} (yellow line) and MLD_CBT^{gb} (purple line) against their baselines. This figure plots the difference in the number of instances solved as a function of time used. At most points in time, MapleLRB^{gb}, MLD^{gb}, and MLD_CBT^{gb} each solves more problems than their baseline. This is particularly pronounced for MLD^{gb} (yellow line) at earlier time points, for MLD_CBT^{gb} (purple line) on mid range time points. The improvement for MapleLRB^{gb} (red line) remains steady, with a brief downward slope in the middle. Glucose^{gb} performs slightly worse than Glucose in between 500 to 1500 seconds, but gains a small advantage later.

Figure 3.4: Solve time comparisons for **TS1**. For any point above 0 in the vertical axis, our extensions solve more instances than their baselines at the time point in the horizontal axis.



PAR-2 Score Comparison

Table 3.3 shows that all our extended versions achieve a lower PAR-2 score than their baselines for all the problem sets. Overall, the percentage of PAR-2 score reductions (computed from the last Column of Table 3.3) with MLD^{gb}, MapleLRB^{gb} and MLD_CBT^{gb} are 3.73%, 2.98% and 2.12%, respectively, which are significant with respect to SAT competition. For example, in SAT Competition 2018 the winning solver was ahead in PAR-2 score by only 0.81%.[2]

Glucose^{gb} also lowers the PAR-2 score over Glucose but by only 0.60%. The improvement is less impressive than with the other three GB extensions. In Section 3.7, we will discuss the reason and show that this performance does not indicate that the GB method is ineffective.

Effectiveness of the GB Method on Benchmark Families

Many benchmarks in **TS1** are of industrial problems. Table 3.4 lists those benchmark families, for which our GB method is particularly efficient. For these benchmark families our GB extended solvers solve at least 2 more instances than their

[2]http://sat2018.forsyte.tuwien.ac.at/

baselines.

Table 3.4: Benchmark families for which the GB extended solvers solve at least two more instances than their baselines. The Column **Baseline** and **GB** show the number of problem solved.

GB Extensions	**Benchmarks/SAT Comp**	**Baseline**	**GB**	**% Improvements**
Glucose^{gb}	Integer Prefix/2017	28	**32 (+4)**	**14.32%**
	Soos/2018	8	**11 (+3)**	**37.5%**
	Ofer/2018	9	**11 (+2)**	**18.18%**
MapleLRB^{gb}	T/2017	28	**31 (+3)**	**9.67%**
	Integer Prefix/2017	27	**30 (+3)**	**10.00%**
	Klieber/2017	17	**19 (+2)**	**10.52%**
	Chen/2018	2	**4 (+2)**	**50.00%**
	Ofer/2018	5	**7 (+2)**	**28.57%**
	Scheel/2018	18	**20 (+2)**	**10.00%**
MLD^{gb}	ak128/2017	11	**13 (+2)**	**15.38%**
	Heule/2018	16	**20 (+4)**	**20.00 %**
MLD_CBT^{gb}	Xiao/2018	7	**9 (+2)**	**22.22%**
	Collatz/2018	7	**10 (+3)**	**30.30%**

3.6 G2L: A New Measure of Solving Efficiency

In [41], Liang 2017 et al. show that on average, better branching heuristics achieve higher average GLR values and lower average LBD (aLBD) scores of their learned clauses. In Table 3.5, we compare our extended solvers and their baselines in these terms. All the solvers with GB extension generate conflicts at about the same rate as their corresponding baselines and achieve slightly smaller average aLBD scores. These results are largely consistent with [41].

Table 3.5: Comparison of average GLR and aLBD score for GB extension solvers and baselines over the 750 test instances.

Systems	Glucose	Glucose^{gb}	MapleLRB	MapleLRB^{gb}	MLD	MLD^{gb}	MLD_CBT	MLD_CBT^{gb}
avg. GLR	0.49	0.49	0.48	0.48	0.40	0.40	0.40	**0.41**
avg. aLBD	20.09	**19.93**	24.88	**24.79**	27.73	**27.36**	27.59	**27.26**

Of course, these are rough measures. One can often find some subset of the benchmarks for which average GLR and average aLBD are not good performance indicators. However, for some subsets of benchmarks it may be highly expected that these metrics should be re-enforced. In this section, we select two subsets of this kind, but surprisingly the standard metrics do not provide a consistent explanation; they even lead to opposite conclusions. However, we show that a simple new measure, based on the fraction of learned clauses that are glue clauses, provides a consistent explanation of solving efficiency.

3.6.1 The G2L Measure

We define a new performance metric called *Glue to Learned* (G2L). Then we present an analysis with all three metrics, aLBD, average GLR and average G2L on two different sets of instances, where the baseline heuristics and their GB extensions show opposite performance.

Definition 2: ***Glue to Learned (G2L)*** *G2L denotes the fraction of learned clauses that are glue clauses. More precisely, it is defined by* $\frac{\#glue\_clauses}{\#learned\_clauses}$, *where our solver* Ψ *has learned #learned_clauses clauses for a given run on a given formula, among which #glue_clauses are glue clauses.*

3.6.2 Relating G2L to Solving Efficiency

The performance of branching heuristics correlates well with average GLR and the average aLBD scores at large scale. However, these two metrics fail to explain the performance of the baseline heuristics and their GB extensions for two specially designed subsets of instances from $\mathbf{TS1}$:

- $\mathbf{GB_{exclusive}}$: Instances solved by Ψ^{gb} but not by Ψ.
- $\mathbf{Baseline_{exclusive}}$: Instances solved by Ψ, but not by Ψ^{gb}.

Table 3.6 compares the four baseline solvers and their GB extensions in terms of average GLR, aLBD, and G2L separately for the two sets $\mathbf{GB_{exclusive}}$ and $\mathbf{Baseline_{exclusive}}$ instances. We expected that the solving efficiency positively (resp. negatively) correlates with average GLR and G2Ls (resp. average aLBD).

We observe the following:

- Average GLR: For instances from $\mathbf{GB_{exclusive}}$ (Column C) and $\mathbf{Baseline_{exclusive}}$ (Column D), the better branching heuristics have lower average GLR values. This is surprising since the performance of branching heuristics is negatively

Table 3.6: Comparison between baselines and their GB extensions for average GLR, average aLBD and average G2L for instance sets $\mathbf{GB_{exclusive}}$ and $\mathbf{Baseline_{exclusive}}$; In Column B, $\{x\}^{gb}$ is the GB extension of baseline heuristic x. Column C (resp. Column D) shows three metrics: avg. GLR, avg. aLBD and avg. G2L for instance category $\mathbf{GB_{exclusive}}$ (resp. $\mathbf{Baseline_{exclusive}}$), where the sub-column #inst shows the number of $\mathbf{GB_{exclusive}}$ (resp. $\mathbf{Baseline_{exclusive}}$) instances for which we are comparing the heuristics in Column B.

(A) Systems	**(B) Employed Heuristics**	**(C) $\mathrm{GB_{exclusive}}$**				**(D) $\mathrm{Baseline_{exclusive}}$**			
		#inst	**avg. GLR**	**avg. aLBD**	**avg. G2L**	**#inst**	**avg. GLR**	**avg. aLBD**	**avg. G2L**
Glucose	{VSIDS}	33	**0.56**	28.60	0.0005	29	0.59	**18.52**	**0.0015**
Glucose^{gb}	{VSIDS}^{gb}		0.53	**24.69**	**0.0016**		**0.62**	20.14	0.00078
MapleLRB	{LRB}	27	**0.50**	26.06	0.00073	14	0.47	**30.75**	**0.00046**
MapleLRB^{gb}	{LRB}^{gb}		0.46	**20.38**	**0.00126**		**0.48**	32.02	0.00037
MLD	{Dist/VSIDS/LRB}	28	**0.55**	**23.60**	0.00029	15	0.53	26.70	**0.0011**
MLD^{gb}	{Dist/VSIDS/LRB}^{gb}		0.51	26.04	**0.00032**		**0.58**	**23.21**	0.0009
MLD_CBT	{Dist,VSIDS,LRB}	26	**0.49**	**26.08**	0.0006	24	0.51	29.64	**0.00065**
MLD_CBT^{gb}	{Dist/VSIDS/LRB}^{gb}		0.43	36.24	**0.0011**		**0.55**	**25.42**	0.00037

correlated with average GLR values. This is highly inconsistent with the results reported in [41].

- Average aLBD: In both $\mathbf{GB_{exclusive}}$ and $\mathbf{Baseline_{exclusive}}$, the better heuristics have lower average aLBD in Glucose and MapleLRB based systems. This is consistent with the results from [41]. However, in MLD and MLD_CBT based systems, the better branching heuristics have higher average aLBD, which is inconsistent with the results of [41].

- Average G2L: For both $\mathbf{GB_{exclusive}}$ and $\mathbf{Baseline_{exclusive}}$, the better heuristics always achieve higher average G2L values. The biggest difference in G2L is 220% (0.0016 - 0.0005) for VSIDS^{gb} in Glucose^{gb} and VSIDS in Glucose for the subset of instances $\mathbf{GB_{exclusive}}$ (Column C). We observe a significantly larger average G2L values for the better solvers in all the other cases as well by comparing the bold values in avg. G2L subcolumn with the values not in bold, for both columns C and D in Table 3.6.

To summarize, for instances for which one heuristic works better than the other, the correlation between the performance of branching heuristics and average GLR and average aLBD is not always consistent with the results of [41]. However, the average value of the new metric G2L positively correlates with the performance in each case.

3.7 Effect of Glue Level Normalization

Earlier, we noticed that Glucose^{gb} showed less improvement than the other GB extensions. Compared to its baseline, Glucose^{gb} solves 4 additional instances, lowers the PAR-2 score only by 0.60% (Table 3.3), and solves instances at a slower rate than its baseline at most time points (Figure 3.4).

Unlike the other 3 baseline solvers used in our experiments, the baseline solver Glucose already bumps variables that are propagated from glue clauses by using VSIDS [8]. These variables are a subset of what we call glue variables. Thus in Glucose^{gb}, these variables get bumped from two sources: from GB bumping and from VSIDS. We hypothesize that the relatively weak performance of Glucose^{gb} comes from this imbalance.

We tested this hypothesis by changing the glue level normalization method in GB to decrease the bumping factor in Alg. 2. For a given glue variable v, instead of dividing $gl(v)$ by $|G|$, we divide by a bigger factor: $\frac{gl(v)}{\sum_{\theta \in G} len(\theta)}$, where $len(\theta)$ is the number of variables in the glue clause θ. The sum is the total number of the glue variables discovered so far in the search. If the average length of the glue clauses in G is n, then in this version, $gl(v)$ is scaled down by n.

We repeated our experiment with this version. Over the 750 instances from $\mathtt{TS1}$, Glucose^{gb} now solves 11 more instances than Glucose and lowers the PAR-2 score by 2.86%. For the other three GB extensions, this reduction does not work well.

3.8 Experiment with Immediate Bumping

We performed a smaller scale experiment with MLD over the 350 instances from SAT competition-2017, where we bump the score of the glue variables as soon as their hosting glue clause is learned (i.e., without delaying the bumping). MLD, with this version of glue variable bumping, solves 2 more UNSAT instances, but 2 fewer SAT instances than the baseline. As this immediate bumping did not appear to be promising with MLD, we did not perform any further experiment.

3.9 Related Work

Glucose [8] increases the activity scores of variables of the learned glue clauses. This bumping is based on the VSIDS score bumping scheme. In contrast, we increase the activity scores of all variables that appear in glue clauses based on their normalized glue level.

In [37], Katsirelos 2012 et al. studied the behavior of Glucose with respect to *eigencentrality*[3], a precomputed static ranking of the variables in industrial SAT instances. The branching and propagated variables in Glucose have a high degree of eigencentrality and compared to the variables in conflict clauses, the variables that appear in learned clauses are more eigencentral. In contrast, we dynamically characterize glue and nonglue variables within the course of a search and show that decisions with glue variables are more conflict efficient than decisions with nonglue variables.

In [45], Liang 2015 et al. show that the VSIDS heuristic branches disproportionately more often on variables that are bridges[4] between communities. Here, we have shown that CDCL heuristics branch disproportionately more often on glue variables.

3.10 Conclusions

In this work, we showed experimentally that decisions with variables appearing in glue clauses are more conflict efficient than decisions with other variables, and that state-of-the-art CDCL SAT solvers tend to make glue decisions more often. Motivated by these observations, we developed a CDCL variable bumping scheme,

[3]Intuitively, *eigencentrality* of a node n in a given graph G measures the importance of n in G, which is computed based on the importance of the neighbours of n.

[4]Given a SAT formula $\mathcal{F}$, one can construct a Variable Incidence Graph (VIG) $G_{\mathcal{F}}$, where two nodes $v, v' \in G_{\mathcal{F}}$ have an edge between them if v and v' both appears in a clause $C \in \mathcal{F}$. For many industrial SAT formulas, their VIG can be decomposed into different communities (as shown in [1]), where a community is a sub-graph that has more internal edges than outgoing edges. In [45], a variable which connects such two sub-graphs/communities is defined as *bridge variables*.

which increases the heuristic score of glue variables based on the frequency of its appearance in glue clauses. Our empirical evaluation showed the effectiveness of the new method on the main track instances from SAT Competition 2017 and 2018 with four state-of-the-art CDCL SAT solvers. For some subsets of SAT Competition 2017 and 2018 benchmarks, our experimental data are surprisingly inconsistent with the standard performance metrics based on GLR and average LBD. We showed that for these subsets of benchmarks, the G2L measure based on the fraction of learned clauses that are glue clauses provides a consistent explanation of our experimental data.

Chapter 4

Conflict Depression Phases in Search using CDCL

4.1 Introduction

In CDCL, a single decision may generate zero or more conflicts. The Global Learning Rate (GLR) [41] measures the number of conflicts per branching decision. State-of-the-art branching heuristics, such as `VSIDS`, `LRB` or `CHB`, have average GLR values of about 0.5, i.e., they produce on average one conflict per two decisions [41].

In this Chapter, we perform an empirical study on the conflict generation pattern in CDCL search. We discover that there are clear patterns of short bursts of conflicts, called *conflict bursts* (CB), followed by longer phases of what we call *conflict depression* (CD), in which the search fails to generate any conflicts for a consecutive number of decisions.

Contributions of this Chapter are as follows:

- We introduce and formulate the notions of CD and CB phases and then present an empirical study of CD and CB phases in CDCL, using the `VSIDS`-based solver glucose 4.2.1 (abbreviated as gLCM)[1] [7] and the `LRB` based solver MapleCOMSPS_Pure_LRB(abbreviated as MplLRB) [44] on the **TS1** test set. Our analysis shows that CD phases occur at a high rate, and often have long duration.

[1]glucose 4.2.1, which implements the Learned Clause Minimization (LCM) [46] technique on top of glucose 4.1.

- We demonstrate a weak correlation between average length of CD phases and solving hardness.
- We study the correlation between the backjumping in CDCL and CD. On average, longer (resp. shorter) CD phases follow longer (resp. shorter) backjumps. However, as the general tendency, longer (resp. shorter) CD phases *occasionally* follow longer (resp. shorter) backjumpings.
- We characterize the new notion of CD as a novel pathological phase for CDCL SAT solving, and present a thorough discussion on various aspects of CD phases, which reveal interesting insights for CDCL SAT solving.

The next section presents relevant notions and definitions for this Chapter.

4.2 Notions and Definitions

Consider a run of a CDCL SAT solver Ψ which makes a total of $d > 0$ decisions. In each decision, a variable is selected according to a branching heuristic. Each decision i ($0 < i \leq d$) leads to some number $c_i \geq 0$ of conflicts.

Definition 3: *(Conflict History) Let $H = \langle c_1, \ldots, c_i, \ldots, c_d \rangle$ be the conflict history of the search up to the d^{th} decision. Here, $c_i \geq 0$ denotes the number of conflicts generated by the i^{th} decision.*

By $\langle c_i \rangle_j^k$, we denote $\langle c_j, \ldots, c_k \rangle$, a sub-sequence of H, where $1 \leq j \leq i \leq k \leq d$.

Definition 4: *(CD Phase) A conflict depression (CD) phase is a sequence of one or more consecutive decisions with no conflict. The subsequence $\langle c_i \rangle_j^k$ is a CD phase if $c_i = 0$ for every i, where $j \leq i \leq k$.*

Definition 5: *(CB Phase) A conflict burst (CB) phase is a sequence of one or more consecutive decisions with at least one conflict. The subsequence $\langle c_i \rangle_j^k$ is a CB phase if $c_i > 0$ for every i, where $j \leq i \leq k$.*

Definition 6: *(Length of CD Phase) The length of a CD phase $\langle c_i \rangle_j^k$ is $len(\langle c_i \rangle_j^k) = k - j + 1$, the number of decisions in that CD phase.*

Example 1: *The conflict history* $H = \langle 1, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, 4, 2, 1, \mathbf{0}, 1, \mathbf{0}, \mathbf{0} \rangle$ *contains 3 CD phases shown in bold:* $\langle c_i \rangle_2^5$, $\langle c_i \rangle_9^9$, *and* $\langle c_i \rangle_{11}^{12}$, *whose lengths are 4, 1 and 2, respectively.* H *also contains 3 CB phases shown in non-bold:* $\langle c_i \rangle_1^1$, $\langle c_i \rangle_6^8$, *and* $\langle c_i \rangle_{10}^{10}$, *whose lengths are 1, 3 and 1, respectively.*

Definition 7: *Suppose the solver* Ψ *takes a total of* d *decisions, performs* p *propagations, makes* r *restarts and encounters* u *CD phases with length* $\langle l_1 \dots l_i \dots l_u \rangle$, *where* l_i *is the length of the* i^{th} *CD phase. We define*

- *the Propagation Rate (PR) as* $\frac{p}{d}$, *the number of propagations per decision.*
- *the Decision Rate (DR) as* $\frac{d}{r}$, *the number of decisions per restart.*
- *the CD phase Rate (CDR) as* $\frac{u}{r}$, *the number of CD phases per restart.*
- *the average CD phase length (aCDPL) as* $\frac{\sum_{i=1}^{u} l_i}{u}$

Let $d' \leq d$ be the number of decisions that generate at least one conflict. Let $d' = d_s + d_m$, where d_s is the number of decisions that generate *exactly one conflict* and d_m is the number of decisions that generate *more than one conflict*.

Definition 8: *We define*

- *The Fraction of Decisions with Conflicts (FDC) as* $\frac{d'}{d}$, *which measures the fraction of decisions that produce at least one conflict. This measure is related to but different from GLR in that it only considers decisions with conflicts, not number of conflicts. We further divide FDC=FDOC+FDMC as follows:*
  - *Fraction of Decisions with One Conflict (FDOC) =* $\frac{d_s}{d}$.
  - *Fraction of Decisions with Multiple Conflicts (FDMC)=* $\frac{d_m}{d}$.

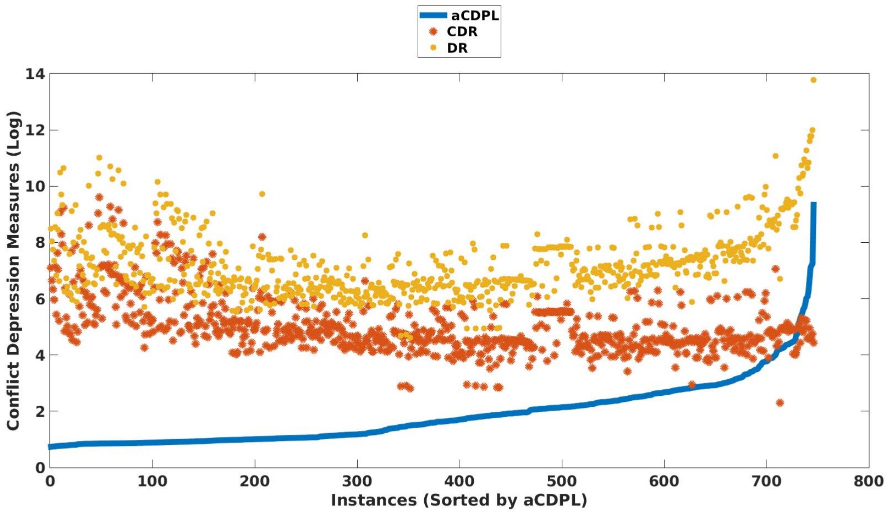
4.3 CD Phases in CDCL Branching Heuristics

We study conflict depression with the `VSIDS` and `LRB` heuristics, and with the CDCL solvers gLCM (which implements `VSIDS` exclusively) and MplLRB (which implements `LRB` exclusively). In **TS1**, 4 instances out of 750 have only a single

CD phase, each of which is exceptionally long. We discarded these 4 instances as outliers, since the aCDPL of these instances are based on only one CD phase.

We collect the following statistics from the search for each remaining instance from **TS1**: DR, CDR, aCDPL, GLR, FDC, FDOC and FDMC.

Figure 4.1: aCDPL, CDR and DR for `VSIDS` in gLCM for **TS1** instances. Each measures are shown in Log (natural) scale.



Conflict Depression in `VSIDS`

Figure 4.1 shows the Decision Rate (DR), CD Phase Rate (CDR) and average CD phase length (on a natural log scale) for instances in **TS1** for `VSIDS`. The instances are sorted by average CD phase length. The average CD phase length (blue line) is short for most instances, but still consists of multiple decisions. Irrespective of their average CD phase lengths, for almost all instances CD phases (orange data points) occur at a high rate given the decision rates (yellow data points).

Figure 4.2 shows the distribution of average lengths of CD phases. This average ranges from 2 to 11,348. 263 instances have a very short length of at most 3. The distribution is heavy-tailed, with 69 instances of average length greater than 25 shown in the rightmost bin in Figure 4.2.

Figure 4.2: Distributions of aCDPL for **TS1** instances for `VSIDS` in gLCM

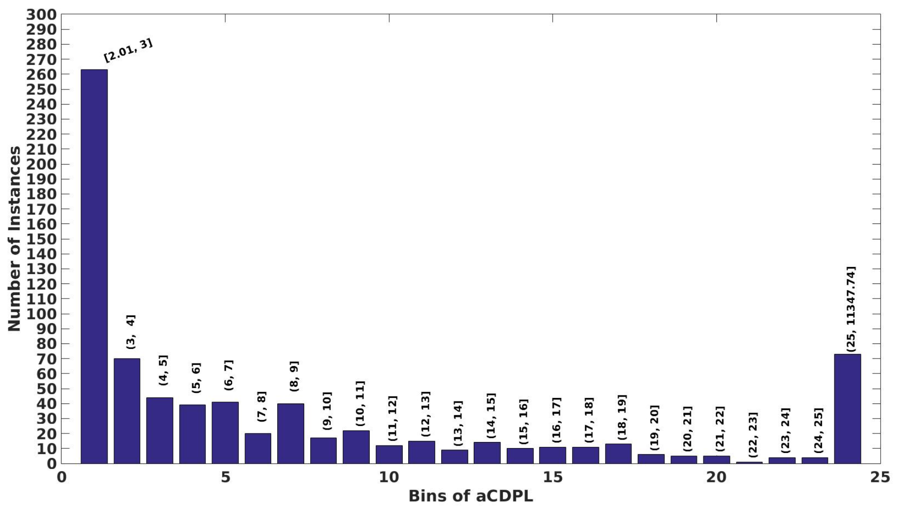


Figure 4.3: aCDPL, CDR and DR for `LRB` in MplLRB for **TS1** instances. Each measures are shown in Log (natural) scale.

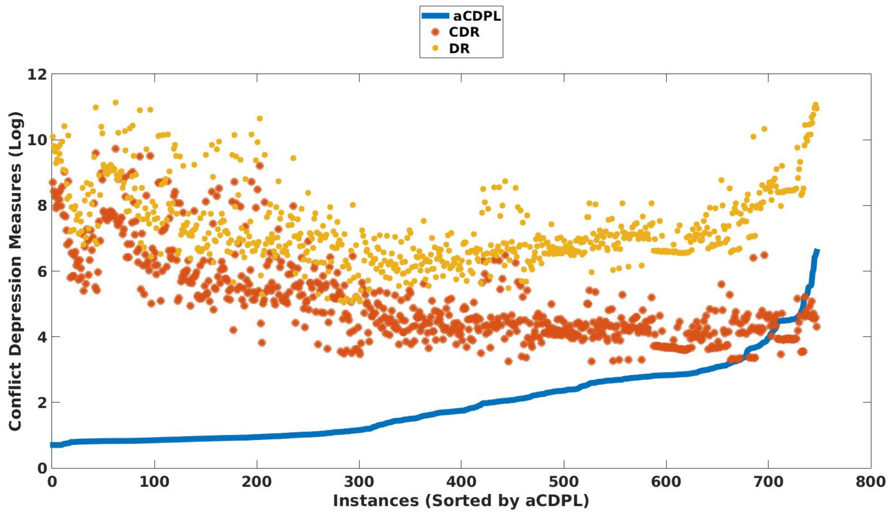
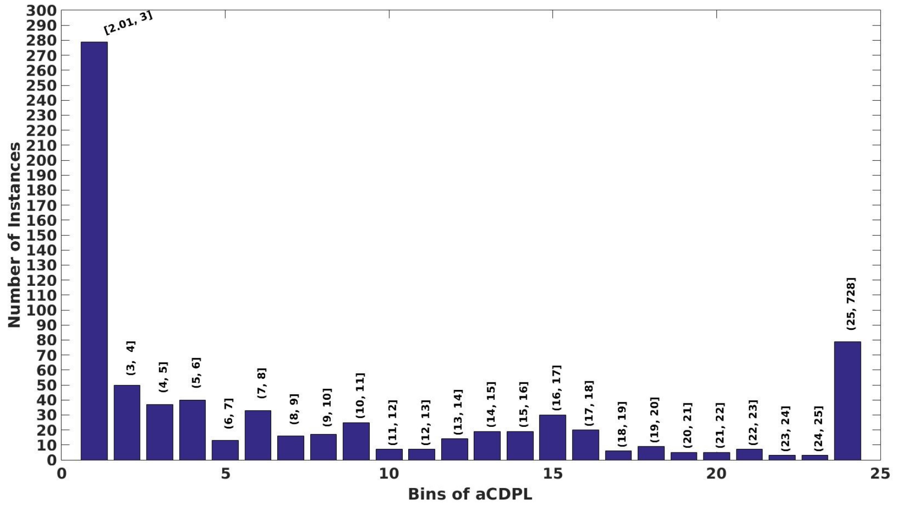


Figure 4.4: Distributions of aCDPL for **TS1** instances for `LRB` in MplLRB



Conflict Depression in `LRB`

Figures 4.3 and 4.4 show the corresponding results for `LRB` in the solver MplLRB for the instances in **TS1**. Similar to `VSIDS`, as shown in Figure 4.3, `LRB` undergoes conflict depression phases at a fast rate with respect to the decision rate. As shown in Figure 4.4, the distribution of instances by CD phase length is also similar to `VSIDS`: a high number (279) of instances have small average CD phase length of at most 3. The distribution is heavy-tailed, with 79 instances of CD phase length greater than 25.

Overall, the data indicates that for both `VSIDS` in gLCM and `LRB` in MplLRB, conflict depressions occur frequently and often last over multiple decisions, leading to a high average CD phase length.

4.3.1 Propagation Depression amid a CD Phase

During a CD phase, the activity scores of `VSIDS` and `LRB` are not a good predictor of a variable's future performance, since branching decisions fail to produce any conflict and perform only truth value propagations. Are there any differences in the pattern of unit propagations between CD and CB phases?

Table 4.1 compares the average PR values over the decisions in CD and CB

Table 4.1: Average Propagation Rate (PR) for CD and CB Phases for `VSIDS` in gLCM

1: Type	**2: #Inst**	**3: Propagation Rate**	
		3.1 CD Phase	**3.2 CB Phase**
SAT	177	153.43	**1560.40**
UNSAT	195	404.40	**3445.40**
Unsolved	378	173.18	**1718.51**
Combined	750	229.51	**2136.73**

Table 4.2: Average PR for CD and CB Phases for `LRB` in MplLRB

1: Type	**2: #Inst**	**3: Propagation Rate**	
		3.1 CD Phase	**3.2 CB Phase**
SAT	189	89.86	**808.61**
UNSAT	186	135.55	**991.95**
Unsolved	375	107.18	**1179.10**
Combined	750	109.82	**1039.40**

phases for `VSIDS`, and Table 4.2 compares the same measures for `LRB`, for the instances from $\mathbf{TS1}$. On average, for both `VSIDS` and `LRB`, the PR values during a CD phase are almost 10 times lower than in CB phases. This demonstrates that during a CD phase, branching decisions of both `VSIDS` and `LRB` go through propagation depression as well.

4.4 Comparing the Average CD Phase Length in gLCM and MplLRB

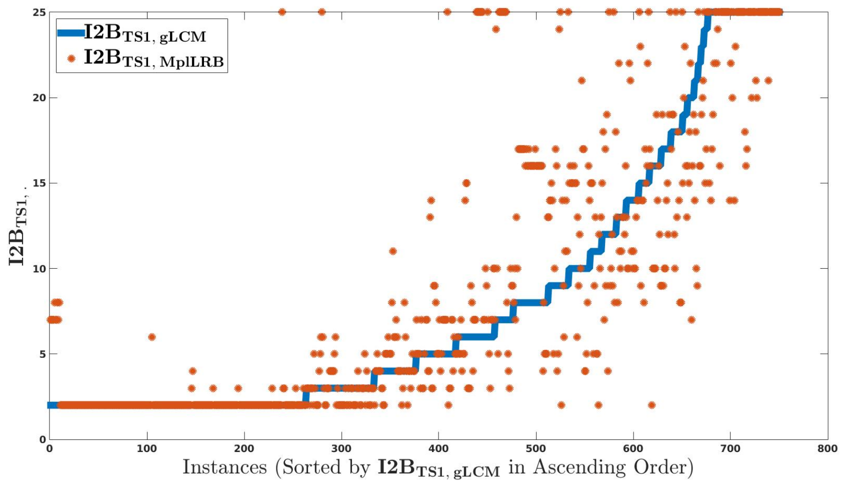
In Figures 4.2 and 4.4 for gLCM and MplLRB, respectively, we showed the distribution of instances from $\mathbf{TS1}$ in 24 bins (starting from bin 2 to 25), where an instance i is put in a bin if the average CD phase length of i falls into the range that the bin represents. Here, we present an analysis to find if these instances change their bins when running with two different solvers.

Let Ψ be a solver and I be a set of instances.We define the set

$$\mathbf{I2B}_{I,\Psi} = \{b \mid b \text{ is the bin for an instance } i \in I, \text{ when running with } \Psi\}$$

which contains the bins for all instances of I, when running with Ψ.

Figure 4.5: Comparing the average CD phase length for gLCM and MplLRB over $\mathbf{TS1}$ instances



We are interested in the change in distribution of instances over bins, when run with different solver on a fixed collection of instances. Figure 4.5 plots $\mathbf{I2B_{TS1,\,gLCM}}$ and $\mathbf{I2B_{TS1,\,MplLRB}}$ for $\mathbf{TS1}$, where the instances are sorted by bin values in $\mathbf{I2B_{TS1,\,gLCM}}$. This plot shows that when solved with MplLRB, about 51.60% of the instances (red dots, which are not on the blue line) in $\mathbf{TS1}$ locate at different bins, compared to their bins with gLCM.

This analysis demonstrates the variation of conflict generation patterns between different solvers over a fixed set of instances in terms of average CD phase length. An instance that has a low average CD phase length with one solver, can have a dramatically high average CD phase length with another solver and vice versa.

4.5 Conflict Bursts with `VSIDS` and `LRB`

How long are the CB phases compared to CD phases? For `VSIDS`, the average length of CB and CD phases over the $\mathbf{TS1}$ instances is 1.67 and 20.63, respectively. For `LRB`, these values are 1.89 and 18.25. For both of these branching heuristics, shorter CB phases are separated by much longer CD phases.

Table 4.3: Average values of GLR, FDC, FDOC and FDMC for `VSIDS` in gLCM

1: Type	2: #Inst	3: GLR	4. FDC	5: FDOC	6: FDMC
SAT	177	0.4644	0.2394	0.0980	0.1414
UNSAT	195	0.5070	0.2492	0.0927	0.1565
Unsolved	378	0.5099	0.2568	0.0992	0.1576
Combined	750	0.4984	0.2507	0.0972	0.1535

Table 4.4: Average values of GLR, FDC, FDOC and FDMC for `LRB` in MplLRB

1: Type	2: #Inst	3: GLR	4: FDC	5: FDOC	6: FDMC
SAT	189	0.4371	0.2294	0.0957	0.1337
UNSAT	186	0.5363	0.2677	0.0958	0.1719
Unsolved	375	0.4899	0.2484	0.0932	0.1531
Combined	750	0.4881	0.2473	0.0945	0.1528

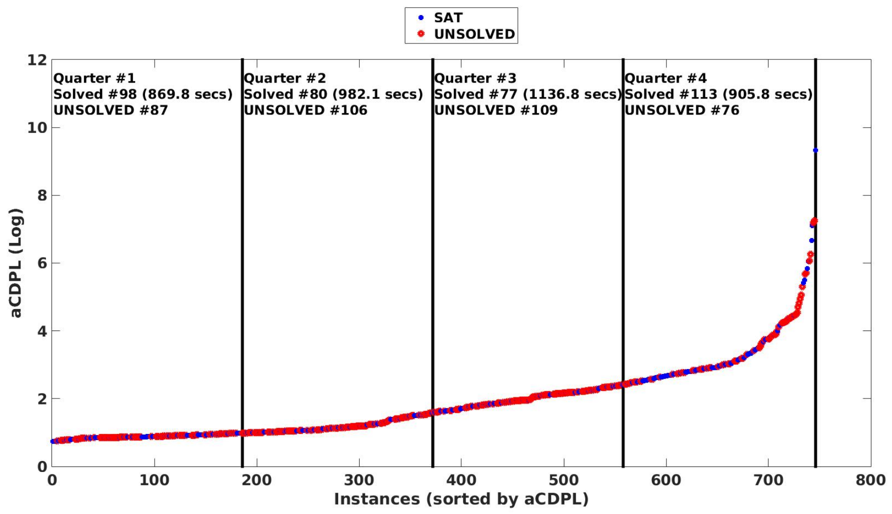
Bursts of Conflict Generation

Tables 4.3 and 4.4 show the average values of GLR, FDC, FDOC and FDMC for `VSIDS` and `LRB`, respectively. The **TS1** instances are grouped into three categories: SAT, UNSAT and Unsolved. Column 3 in both tables shows that the average GLR values for all three types of problems are close to 0.5. In contrast, the average FDC values in Column 4 in both tables are much lower, close to 0.25. Only 25% of all the decisions produce at least one conflict. The majority of the conflict producing decisions produce more than 1 conflict: FDMC (Column 6 of Table 4.3 and 4.4) is much larger than FDOC (Column 5 of Table 4.3 and 4.4) in all three categories.

We have the following conclusion:

- The typical search behavior of CDCL contains short CB phases followed by longer CD phases, in which the search does not find any conflicts.
- During a CD phase, the search goes through propagation depression as well.
- The short CB phases are conflict intense: within a few decisions, many conflicts are generated.

Figure 4.6: Performance of gLCM on **TS1** instances and their average CD phase length



4.6 Average CD Phase Length and Performance of Solvers

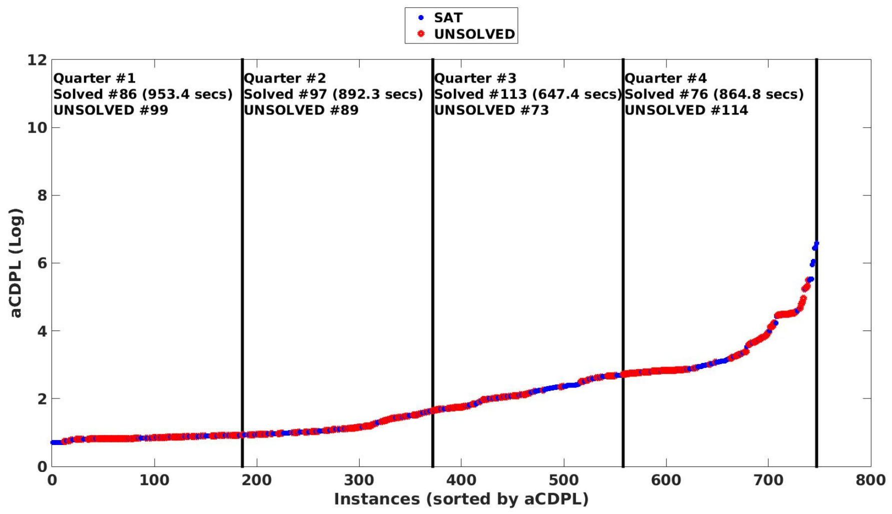
Our analysis shows that for many **TS1** instances, the average CD phase length (aCDPL) is very high for both gLCM and MplLRB. Since a faster rate of conflict discovery (high GLR) is correlated with solving efficiency [41], the question arises: How does CD correlate with the performance of CDCL solvers?

In this section, we study if there is any correlationship between solving hardness of CDCL solvers for a fixed set of benchmarks and their average CD phase length for the instances of that fixed set of benchmarks. We use gLCM and MplLRB as the CDCL SAT solvers and **TS1** as the test sets.

4.6.1 Grouping of Instances by Increasing CD Phase Length

Figures 4.6 and 4.7 show aCDPL (in natural log scale) for **TS1** instances with gLCM and MplLRB, respectively. The instances are sorted by their aCDPL and are divided into four quarters. In both figures, points representing solved instances are colored in blue and points representing unsolved instances are colored in red.

Figure 4.7: Performance of MplLRB on $\mathbf{TS1}$ instances and their average CD phase length



Observations for gLCM

For gLCM (Figure 4.6),

- The second (106) and third (109) quarter have the most unsolved instances.
- The instances solved in these two quarters have higher solving time than the other two quarters on average.
- Among these four quarters, the first and the fourth quarters have instances with lowest and highest average CD phase length, respectively. Compared to the second and the third quarters, where average CD Phase length is moderate-to-high, the first and fourth quarter have more solved instances with lower solve time.

Hence, for gLCM, many instances with very low (quartile 1) and very high (quartile 4) CD phases are easier to solve than instances with moderate-to-high (quartile 2 and 3) average CD phase length.

Observations with MplLRB

For MplLRB, we observe the opposite behavior with respect to aCDPL for these instances. In Figure 4.7, we see that

- more unsolved instances are in the first and last quarters, where instances have lowest and highest average CD phase lengths. There are more hard instances in the first (instances with low aCDPL) and last quarters (instances with very high aCDPL).

From this analysis we cannot draw any general conclusions on how aCDPL correlates to a solver performance.

4.6.2 Grouping of Instances by Benchmark

We further analyzed the performance of solvers with respect to aCDPL for a set of selected benchmarks from $\mathbf{TS1}$. We compare the aCDPL of solved and unsolved instances to relate solving hardness with aCDPL. We performed this analysis for both gLCM and MplLRB and observed similar results.

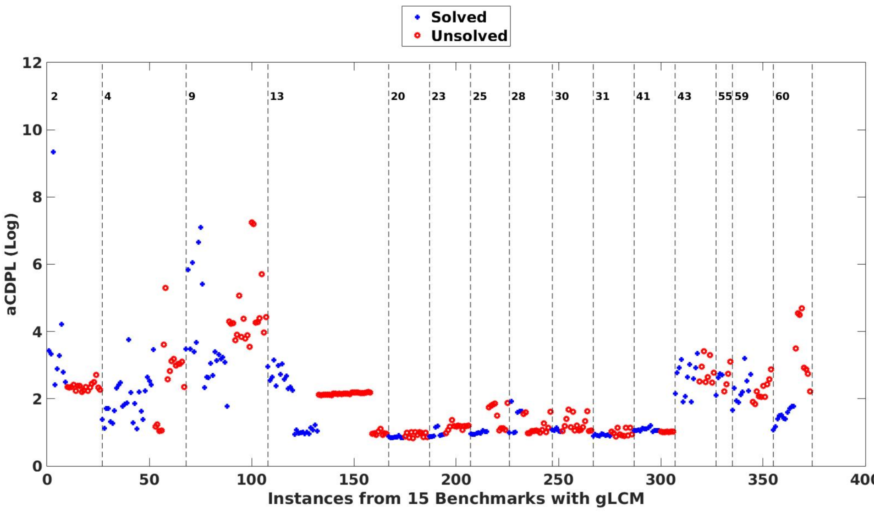
We identified 61 different benchmarks in $\mathbf{TS1}$. For each benchmark β and solver ψ, we categorize the instances of β into two types:

- $\mathbf{E}_{\psi^\beta}$: easy instances in benchmark β are solved by ψ before timeout.
- $\mathbf{H}_{\psi^\beta}$: hard instances in benchmark β, which ψ is unable to solve before timeout.

For a benchmark β and a solver ψ, if $|\,\mathbf{E}_{\psi^\beta}|$ or $|\,\mathbf{H}_{\psi^\beta}|$ is too small, then the comparison of aCDPL between $\mathbf{E}_{\psi^\beta}$ and $\mathbf{H}_{\psi^\beta}$ will not be robust. Therefore, we consider only those benchmarks for which both $\mathbf{E}_{\psi^\beta}$ and $\mathbf{H}_{\psi^\beta}$ contain more than 25% of the instances in β.

- Out of 61 benchmarks, only 15 are robust for gLCM: $|\mathbf{E}_{\mathtt{gLCM}^\beta}| > |\beta| * 0.25$ and $|\mathbf{H}_{\mathtt{gLCM}^\beta}| > |\beta| * 0.25$.
- Out of these 61 benchmarks, only 15 are robust for MplLRB: $|\mathbf{E}_{\mathtt{MplLRB}^\beta}| > |\beta| * 0.25$ and $|\mathbf{H}_{\mathtt{MplLRB}^\beta}| > |\beta| * 0.25$.

Figure 4.8: aCDPL of gLCM on $\mathbf{TS1}$ instances and their average CD phase length



Here $|\beta|$ denotes the number of instances in the benchmark β. Given a set of robust benchmarks for ψ, we define its two subsets:

- $\mathbf{H^{\psi}_{hg}}$: a set of robust benchmarks for which the median values of aCDPL for unsolved instances with ψ are **higher** than solved instances.
- $\mathbf{H^{\psi}_{lw}}$ a set of robust benchmarks, for which the median values of aCDPL for unsolved instances with ψ are **lower** than the solved instances.

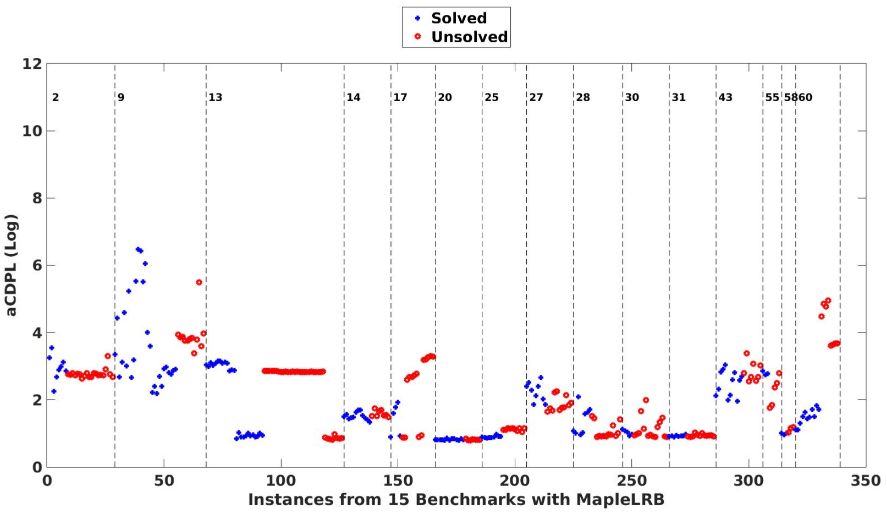
Observations with gLCM

Figure 4.8 shows the aCDPL of the instances from 15 benchmarks which are robust for gLCM. Each of these benchmark are identified by their Benchmark ID (or Bench ID, shown in top in black numbers) and are separated by vertical dotted bars. Here, solved instances are colored in blue and unsolved are colored in red.

Our observations are:

- There are few benchmarks for which solved instances have higher aCDPL than unsolved instances (for example, Benchmarks with Bench ID 2 and 13).
- For some other benchmarks, all the unsolved instances have higher aCDPL

Figure 4.9: aCDPL of MplLRB on $\mathbf{TS1}$ instances and their average CD phase length



than unsolved instances (for example, Benchmarks with Bench ID 25 and 60).

Table 4.5 compares the median aCDPL values of solved (Column I) and unsolved (Column II) instances for both $\mathbf{H}_{\mathbf{hg}}^{\mathrm{gLCM}}$ (top 8 entries) and $\mathbf{H}_{\mathbf{lw}}^{\mathrm{gLCM}}$ (bottom 7 entries) for gLCM. We have the following observation:

- The median of a slight majority of the robust benchmarks (8/15) for gLCM correlate well with its solving hardness.

Observations with MplLRB

The results for MplLRB are shown in Figure 4.9 and Table 4.6. We have similar observations with MplLRB as gLCM.

To summarize, the median of the measure aCDPL has positive correlation with solving hardness for a slight majority of the benchmarks that we have considered. Hence, there is a weak correlation between aCDPL and solving hardness.

Table 4.5: Comparison of median of aCDPL for solved and unsolved instances of 15 robust benchmarks from $\mathbf{TS1}$ with gLCM; The top 8 rows show the results for $\mathbf{H}_{\mathbf{hg}}^{\mathrm{gLCM}}$ and the bottom 7 rows show the results for $\mathbf{H}_{\mathbf{lw}}^{\mathrm{gLCM}}$ instances.

Entry	**Bench ID**	**Instance Count**	**Solved**		**Unsolved**	
			Instance Count	**I: Median aCDPL**	**Instance Count**	**II: Median aCDPL**
1	4	41	26	6.35	15	19.96
2	9	40	21	27.43	19	70.63
3	20	20	8	2.35	12	2.6
4	23	20	8	2.5	12	3.27
5	25	19	9	2.66	10	5.1
6	30	20	5	2.89	15	3.19
7	31	20	9	2.5	11	2.58
8	60	19	11	4.44	8	25.83
9	2	26	9	26.54	17	10.47
10	13	59	25	9.48	34	8.52
11	28	21	7	4.9	14	2.86
12	41	20	13	2.9	7	2.78
13	43	20	12	15.06	8	15.01
14	55	8	4	14.49	4	13.49
15	59	20	10	9.18	10	8.54

4.7 Correlation of CD Phases and Backjumping Length

Consider a run of a given CDCL solver with a given instance. After learning a clause from the latest conflict and backjumping to a previous level, the search continues by assigning the opposite of the fUIP literal of that last conflict. After backjumping, if the search enters into a space, which is different than the space it was in before the backjumping, then the variable ranking of the currently active decision heuristic may be an inaccurate estimation of the conflict generation potential, and may lead to conflict depression. Based on this intuition, here we study the correlation between conflict depressions and backjumps.

Assume a run of a given CDCL solver with a given instance.

- Following the discovery of a conflict and learning of a clause, assume that the search has just backjumped to level b from the current level c. We define the *backjumping length* of this backjump as $bl = c - b$. Assume that during the whole run, the search finds n conflicts and backjumps n times. We define average of Search Backjumping Length or $\overline{\mathbf{sbl}}$ as the average of bl over those

Table 4.6: Comparison of median of aCDPL for solved and unsolved instances of 15 robust benchmarks from $\mathbf{TS1}$ with MplLRB; The top 8 rows shows the results for $\mathbf{H}_{\mathbf{hg}}^{\text{MplLRB}}$ and bottom 7 rows shows the results for $\mathbf{H}_{\mathbf{lw}}^{\text{MplLRB}}$ instances.

Entry	**Bench ID**	**Instance Count**	**Solved**		**Unsolved**	
			Instance Count	**I: Median aCDPL**	**Instance Count**	**II: Median aCDPL**
1	9	39	27	20.27	12	45.68
2	14	20	12	4.44	8	4.67
3	17	19	5	4.95	14	14.95
4	25	19	9	2.43	10	3.09
5	31	20	8	2.52	12	2.54
6	43	20	12	13.26	8	15.36
7	58	6	3	2.73	3	3.16
8	60	19	11	4.47	8	63.95
9	2	28	8	18.95	20	15.64
10	13	59	25	17.54	34	16.95
11	20	20	13	2.25	7	2.23
12	27	20	9	9.84	11	5.94
13	28	21	8	3.9	13	2.59
14	30	20	5	2.85	15	2.63
15	55	8	3	16.15	5	10.57

n backjumps.

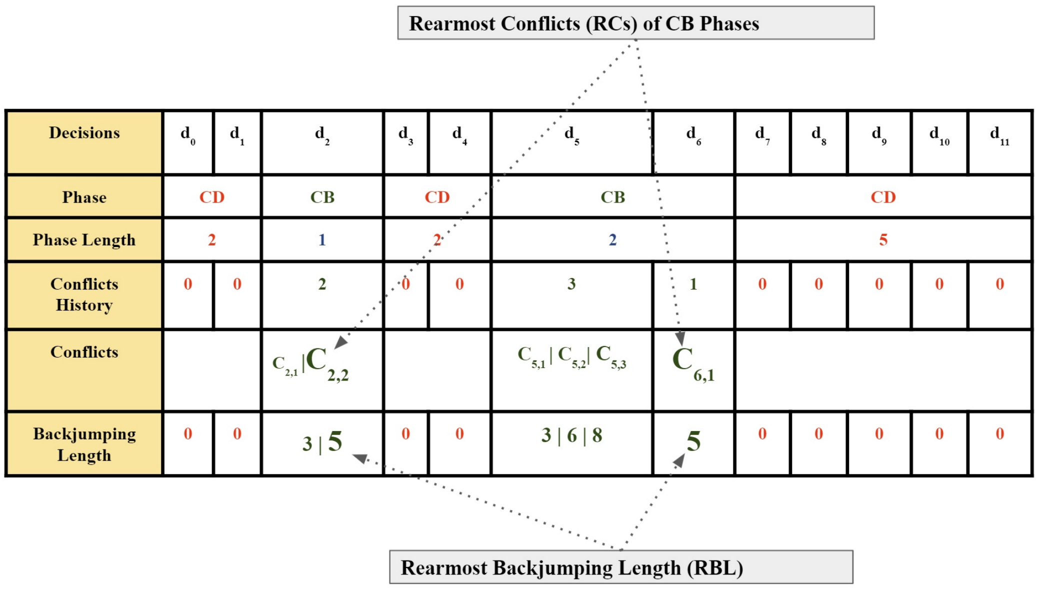
- Assume that the search has just finished a CB phase. We denote the last conflict in a CB phase as the *rearmost conflict* (RC) for that CB, the backjumping that follows RC as the *rearmost backjumping* (RB), and the backjumping length of the RB as the *rearmost backjumping length* (RBL). The RB of a CB phase is followed by a CD phase. In Figure 4.10, the conflicts $\mathbf{C_{2,2}}$ and $\mathbf{C_{6,1}}$ are the RCs of the CB phases that start at decisions d_2 and d_5, respectively, both with RBL of 5. The CB phases are followed by CD phases of length 2 and 5, respectively.

  Let the search encounter m CB and m CD phases. These m CB phases have m RCs. We define $\overline{\mathbf{rbl}}$ as the average backjumping length of those m RCs, each of which is followed by a CD phase.

4.7.1 Longer Backjumps before CD Phases

Figure 4.11 compares $\overline{\mathbf{sbl}}$ (red), the average backjumping length over all conflicts during a run and $\overline{\mathbf{rbl}}$ (blue), the average rearmost backjumping length over the

Figure 4.10: Examples of Rearmost Conflicts and Rearmost Backjumping Length


Rearmost Conflicts (RCs) of CB Phases
Rearmost Backjumping Length (RBL)

Decisions	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}
Phase	CD		CB	CD		CB		CD				
Phase Length	2		1	2		2		5				
Conflicts History	0	0	2	0	0	3	1	0	0	0	0	0
Conflicts			$C_{2,1}$ | $C_{2,2}$			$C_{5,1}$ | $C_{5,2}$ | $C_{5,3}$	$C_{6,1}$					
Backjumping Length	0	0	3 | 5	0	0	3 | 6 | 8	5	0	0	0	0	0

conflicts for instances from $\mathbf{TS1}$ with gLCM (left plot) and MplLRB (right plot) in natural log scale. For both solvers, for almost all of the instances, $\overline{\mathbf{rbl}}$ is larger than $\overline{\mathbf{sbl}}$. Hence, on average a CD phase follows a conflict for which the backjumping length is higher than average.

4.7.2 Correlation between length of CD phase and RBL

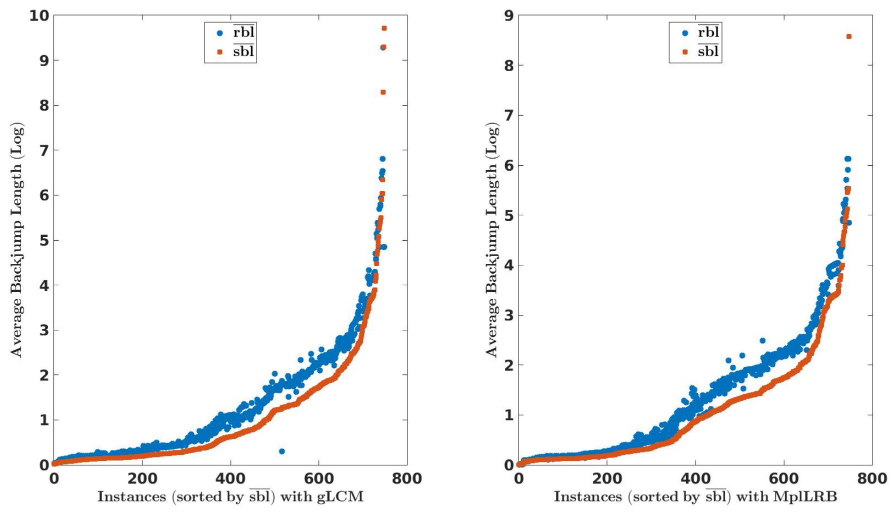
Is a longer RB in a CB phase followed by a longer CD phase? In this section we study this question.

Let $\mathbf{L} = \langle \mathbf{lcd}_1, \ldots, \mathbf{lcd}_i, \ldots, \mathbf{lcd}_m \rangle$ be the length of m CD phases. Let $\mathbf{B} = \langle \mathbf{rbl}_1, \ldots, \mathbf{rbl}_i, \ldots, \mathbf{rbl}_m \rangle$ be the sequence of m RBLs, where $\mathbf{rbl}_i$ is the RBL of the RB, which is followed by the CD phase with length $\mathbf{lcd}_i$.

Let $\rho_{\mathbf{B},\mathbf{L}}$ be the Pearson Correlation Coefficient [36] between $\mathbf{B}$ and $\mathbf{L}$ for a run of a given instance with a given solver. Figure 4.12 shows the $\rho_{\mathbf{B},\mathbf{L}}$ values for $\mathbf{TS1}$ instances with gLCM (left plot) and MplLRB (right plot). In both of these plots, we divide the instances into four types (similar to the categorization in [59]) based on their correlation. Instances with

- **negative** correlation ($\rho < 0$) are marked as red cross.

Figure 4.11: Comparisons between search backtracking length $\overline{\mathbf{sbl}}$ and $\overline{\mathbf{rbl}}$ for $\mathbf{TS1}$ instances.



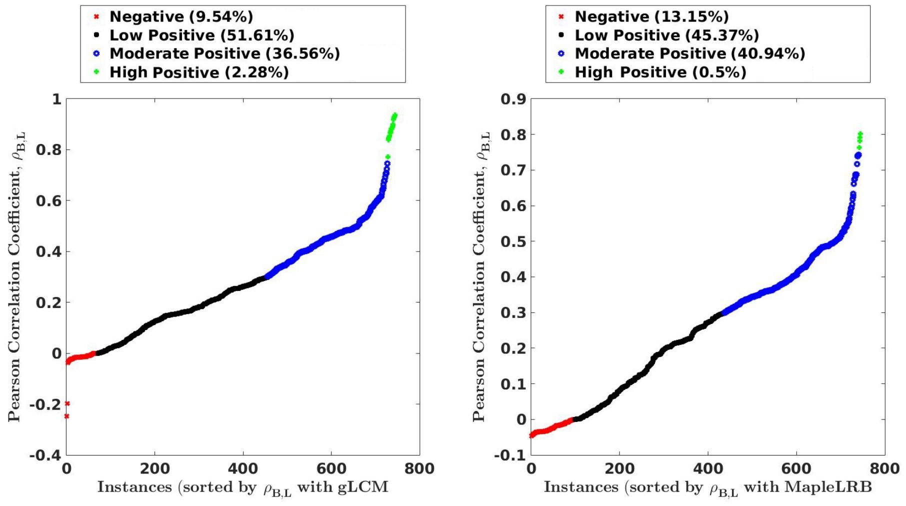
- **low positive** correlation ($0 < \rho < 0.29$) are marked as black stars.
- **moderate positive** correlation ($0.30 \leq \rho \leq 0.75$) are marked as blue circles.
- **high positive** correlation ($\rho \geq 0.75$) are marked as green pluses.

Our observations are as follows. For both solvers,

- A small percentage (9.47% for gLCM and 13.15% for MplLRB) of $\mathbf{TS1}$ instances have small negative $\rho_{\mathbf{B},\mathbf{L}}$ values (red crosses).
- For the majority of instances, the value of $\rho_{\mathbf{B},\mathbf{L}}$ is low (black stars) to moderately (blue circles) positive.
- Only a very small percentage of the instances have high positive correlation (green pluses).

The low to moderate positive values $\rho_{\mathbf{B},\mathbf{L}}$ for the majority of the instances indicate that during the course of the search with these instances, occurrences of longer (resp. shorter) CD phases *occasionally* follow longer (resp. shorter) rearmost backjumps. Now, during the course of a search, many rearmost backjumps can have the same length, for which their corresponding CD phase length can vary. A variation

Figure 4.12: The correlation $\rho_{\mathbf{B},\mathbf{L}}$ between the sequences $\mathbf{B}$ and $\mathbf{L}$ for $\mathbf{TS1}$ instances with gLCM (left plot) and MplLRB (right plot)



in CD phase lengths for a fixed RBL could explain the low-to-moderate positive value of $\rho_{\mathbf{B},\mathbf{L}}$.

Next, we study the correlation between the *average* length of CD phases that follow the rearmost backjump of a fixed length, and that fixed length.

Assume that out of m RBLs, m' RBLs are unique. Let

$$\mathbf{U} = \langle \mathbf{rbl_1}, \ldots, \mathbf{rbl_i}, \ldots, \mathbf{rbl_{m'}} \rangle$$

be the tuple of those unique m' RBLs.

For any RBL, $\mathbf{rbl} \in \mathbf{U}$, we define $\overline{\mathbf{lcd_{rbl}}}$ as the average length of CD phases that follows the RBs with length $\mathbf{rbl}$. For example, In Figure 4.10 two RCs with RBL 5 are followed by 2 CD phases of lengths 2 and 5, respectively. In this example $\overline{\mathbf{lcd}}_5 = \frac{2+5}{2} = \frac{7}{3} = 3.5$.

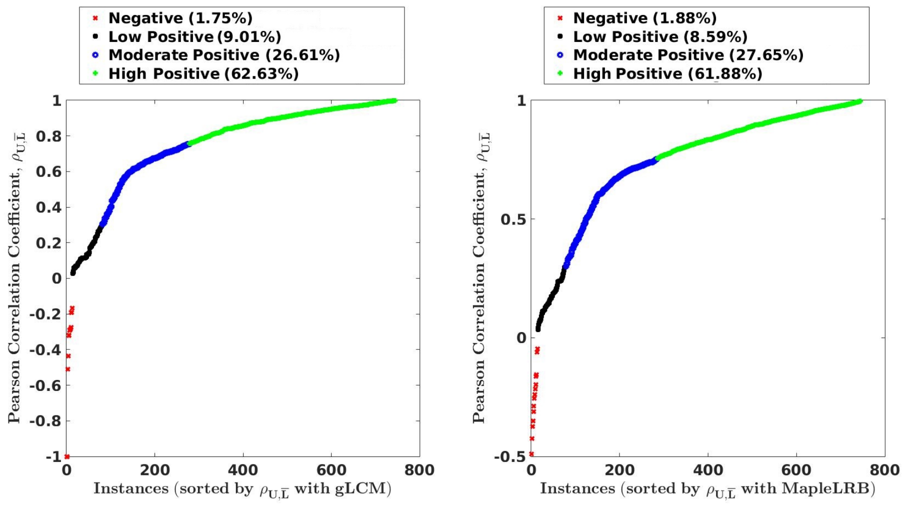
Assume another tuple

$$\overline{\mathbf{L}} = \langle \overline{\mathbf{lcd_1}}, \ldots, \overline{\mathbf{lcd_i}}, \ldots, \overline{\mathbf{lcd_{m'}}} \rangle$$

where $\overline{\mathbf{lcd_i}}$ is the average length of CD phases that follows the RBs with length $\mathbf{rbl_i}$.

Figure 4.13 shows the Pearson Correlation Coefficient $\rho_{\mathbf{U},\overline{\mathbf{L}}}$ between $\mathbf{U}$ and $\overline{L}$ for $\mathbf{TS1}$ instances with gLCM (left plot) and MplLRB (right plot). For both solvers,

Figure 4.13: The correlation $\rho_{\mathbf{U},\overline{\mathbf{L}}}$ between $\mathbf{U}$ and $\overline{\mathbf{L}}$ for $\mathbf{TS1}$ instances with gLCM (left plot) and MplLRB (right plot).



- A very small percentage (1.75% for gLCM, 1.88% for MplLRB) of the instances have negative correlation $\rho_{\mathbf{U},\overline{\mathbf{L}}} < 0$ (red cross).
- A higher percentage of instances (9.01% for gLCM and 8.59% for MplLRB) have low positive correlation $\rho_{\mathbf{U},\overline{\mathbf{L}}} \leq 0.29$ (black stars).
- The high majority of instances have moderate (blue circles, $0.30 < \rho_{\mathbf{U},\overline{\mathbf{L}}} \leq 0.75$) to high (green pluses, $\rho_{\mathbf{U},\overline{\mathbf{L}}} \geq 0.75$) positive correlation between $\mathbf{U}$ and $\overline{\mathbf{L}}$.

Hence, **on average**, longer CD phases are followed by longer backjumps.

Summary of observations

We summarize our observations as follows:

(i) On average, the length of rearmost backjumps (which are followed by CD phases) is higher than average (Figure 4.11).

(ii) **Observations about** $\rho_{\mathbf{B},\mathbf{L}}$ **:** For a good majority of the $\mathbf{TS1}$ instances (around 50% for both solvers), the correlation between backjumping length of rearmost conflict and length of CD phase that follows the conflict is low-to-moderate positive (Figure 4.12). For the large majority of instances, the length of rearmost backjumps are weakly correlated with the length of its CD phase.

(iii) **Observations about $\rho_{\mathbf{U},\overline{\mathbf{L}}}$:** When CD phases are grouped by the length of the RB, which the CD phases follow, the correlation between the average length of CD phases in a group, and the length of the rearmost backjump of that group is high. On average, longer CD phases are followed by longer rearmost backjumps.

4.8 Longer CD Phases Occur Just after a Restart

Gomes et al. [29] observed that the distribution of solution time for a given instance with DPLL-like solvers is *heavy-tailed* under the assumption of a randomized branching heuristic. A heavy tail indicates that mistakes occur frequently in early branching decisions. Initially, restarts in DPLL based solvers were employed as an effective way to combat these early mistakes in branching decisions. In state-of-the-art CDCL SAT solvers, restarts function as a mechanism to rescue the search from an area, where the search is not learning useful clauses [4].

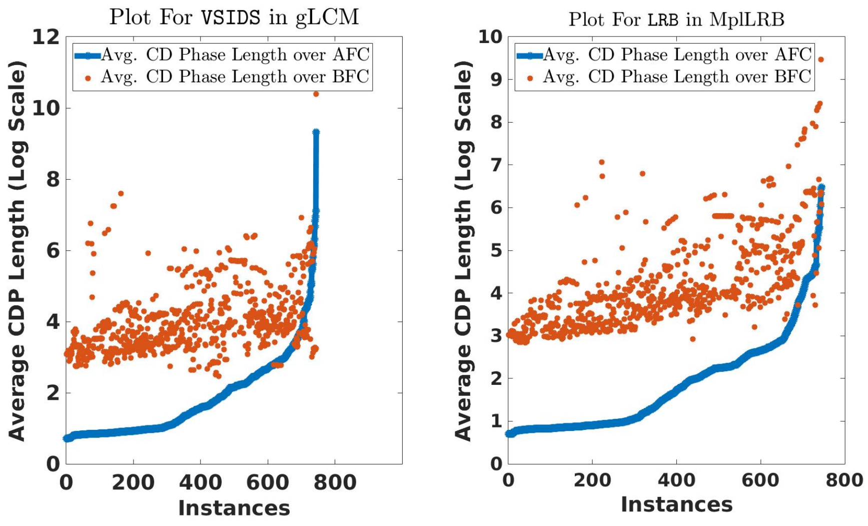
A CDCL solver usually performs many *restarts* during a run, where it abandons the current partial assignment and starts the search from scratch. However, other information such as activity scores and saved phases are preserved. Hence after a restart, the search moves back near the area where it was searching before the restart. The question arises if CDCL search undergoes a longer CD phase right after a restart, compared to later CD phases following the first conflict after the restart. We analyze this question with gLCM and MplLRB for the instances from $\mathbf{TS1}$.

Suppose that a given run of solver Ψ performs n restarts: $r_1 \ldots r_n$. In the search period between restarts r_i and r_{i+1}, we distinguish two types of CD phases as follows:

- **Before First Conflict (BFC) CD Phase:** The single CD phase that occurs before the first conflict following restart r_i.

- **After First Conflict (AFC) CD Phases:** All the other CD phases in the same period.

Figure 4.14: Comparing BFC and AFC CD Phases. Longer CD phases occur right after restarts. Instances are sorted by average CD phase length over AFC CD phases



We run gLCM and MplLRB for each instance of **TS1** until the instance is solved, or the search encounters 10,000 restarts, or 1,000 seconds CPU time is reached. For each run we separately collect: Average CD phase length in BFC and AFC CD phases. The left and right plots of Figure 4.14 compare these numbers for VSIDS (in gLCM) and LRB (in MplLRB). The plots are in Log scale, with instances sorted by AFC. The figure clearly shows that for the overwhelming majority (95.56% for VSIDS and 99.04% for LRB) of instances, the average CD phase length in AFC (blue lines in Figure 4.14) is significantly lower than in BFC (orange dots in Figure 4.14) for both VSIDS and LRB.

On average, search undergoes much larger CD phases right after a restart.

4.9 CD Phase: a Pathological Phase for CDCL SAT Solving

Heuristics guided search algorithms often exhibit *pathological* phases. For example, local search for SAT frequently encounters movement through *plateau regions*, where a sequence of states that the search visits are of equal heuristic value [27].

Similar pathological behaviors occur in deterministic planning, where heuristic values of states do not improve within a plateau region [33].

While the CDCL branching heuristics such as `VSIDS` and `LRB` have different heuristic values for each selected variable amid a CD phase, none of these selected variables generate any conflicts. In this sense, the lack of progress of CDCL SAT search amid a CD phase is similar to the lack of progress of local search algorithms in a plateau region. However, in some special cases, occurrence of substantially long CD phases may be useful to achieve search efficiency. For example, for UNSAT instances, sometimes it may be useful for search to undergo a substantially long CD phase to find a conflict which is relevant for the proof of unsatisfiability. While occurrences of such substantial CD phases are important for constructing a proof, in general CD phases are unproductive.

Therefore, we characterize CD as a pathological phase for CDCL.

4.10 Conclusions

In this Chapter, we have formulated two novel notions for CDCL SAT search: conflict depression phases, a sequence of decisions that do not lead to any conflict and conflict burst phases, a sequence of decisions with at least one conflict each. With two high-performance CDCL SAT solvers and recent SAT competition benchmarks, we have shown empirically that the typical CDCL search behavior consists of short but intense conflict burst phases, followed by longer conflict depression phases.

We have demonstrated that there is a weak correlation between average CD phase length and solving hardness. We have presented an empirical study that reveals correlation between backjump length and CD. We also showed that on average, a CDCL solver undergoes longer CD phases just after restarts. Finally, we characterize CD as a pathological phase for CDCL SAT search.

In the next chapter, we present an algorithmic extension of CDCL that performs randomized exploration amid substantial CD phases, with the goal of swift escape from those CD phases.

Chapter 5

Guiding CDCL via Exploration amid Conflict Depression

During a CD phase, the search does not produce any conflict for a sequence of consecutive decisions. In a substantial CD phase, the lack of conflicts is problematic for the search.

- It spends a considerable amount of resources to perform a sequence of decisions and unit propagations, without learning any clauses that could lead to future pruning. In Chapter 4, we have observed a weak positive correlation between average CD phase length and solving hardness.

In this Chapter, we propose our new CDCL solver extension `expSAT` to perform random exploration amid substantial CD phases. This extension applies random walks amid *substantial* CD phases to learn an updated variable selection heuristic, with the goal of escaping from CD phases quickly. Such exploration visits possible *future* search states, while the standard CDCL branching heuristics rely on conflicts generated from *past* search states.

The SAT community was one of the first to embrace randomized exploration methods, which are a key ingredient of the local search methods of GSAT [71] and WalkSAT [70]. Modern CDCL SAT solvers also utilize exploration methods. Examples are random variable selection in the solver MiniSAT [23] and the trigger of local search episodes at regular intervals in the solver CaDiCal [12]. Randomized exploration was shown to be useful in many related search paradigms. For example, exploration can potentially make a search more robust by mitigating "early mistakes" caused by inaccurate heuristics [82]. Examples of exploration methods are Monte Carlo Tree Search (MCTS) [14] and random walk techniques, which have been successfully applied to both classical [51, 81] and motion planning [39]. MCTS-based search methods were also shown to be effective in automated theorem proving [24] and in puzzle domains [61]. Perhaps the best-known example which combines many of the recent advances in both MCTS and machine learning of heuristics is the super-human strength Go-playing program *AlphaGo* [73].

The contributions of this Chapter are:

1. A formulation of the `expSAT` algorithm, an exploration-driven extension of CDCL. As soon as a substantial CD phase is detected, an `expSAT` solver performs random exploration to identify more conflict potent variables for branching. The heuristic score of variables that cause conflicts during exploration is boosted. We extend both `VSIDS` and `LRB` with this technique.

2. An extensive empirical evaluation of `expSAT` implemented on top of five state-of-the-art SAT solvers, gLCM [7], MplLRB [44], MapleCOMSPS (MplCOMSPS) [44], Maple_CM (MplCM) [65], and MapleLCMDist_ChronoBT (MplCBT) [65]. Among these five solvers, gLCM implements `VSIDS` exclusively and MplLRB uses `LRB` exclusively. The other three solvers employ a combination of `VSIDS` and `LRB`, which are used at different states in their search. Our implementation resulted in a total of 11 `expSAT` extended solvers.

   We perform three sets of experiments using two instance sets: $\mathbf{TS1}$ and $\mathbf{TS2}$ (See Chapter 2 for the details of $\mathbf{TS1}$ and $\mathbf{TS2}$).

   i. In the first set of experiments, we evaluate the `expSAT` extension of `VSIDS` in gLCM, MplCOMSPS, MplCM, and MplCBT.

   ii. In the second set of experiments, we evaluate the `expSAT` extension of `LRB` in MplLRB, MplCOMSPS, MplCM and MplCBT.

   iii. In the last set of experiments, we evaluate a combination of the `expSAT` extensions of `VSIDS` and `LRB` in MplCOMSPS, MplCM, and MplCBT.

Summary of Results: On the benchmarks from the SAT-2017 and SAT-2018 maintrack (**TS1**), most `expSAT` extensions (9/11) solve more instances and achieve a lower (better) PAR-2 score than their respective baseline. The best performing `expSAT` solver solves 16 more instances than its baseline, which is a strong performance gain. On the 52 hard instances from SATCoin cryptographic benchmarks (**TS2**), most of our `expSAT` extensions show very strong gains over their respective baselines.

3. An analysis of the experimental results shows that our results are consistently explained by two standard performance metrics, GLR and average LBD [41]. We also show that exploration reduces the average length of CD phases in instances where `expSAT` is effective.

4. An adaptive algorithm to update exploration parameters during the search and an experimental comparison of this adaptive version of `VSIDS` extension of `expSAT` with the non-adaptive one.

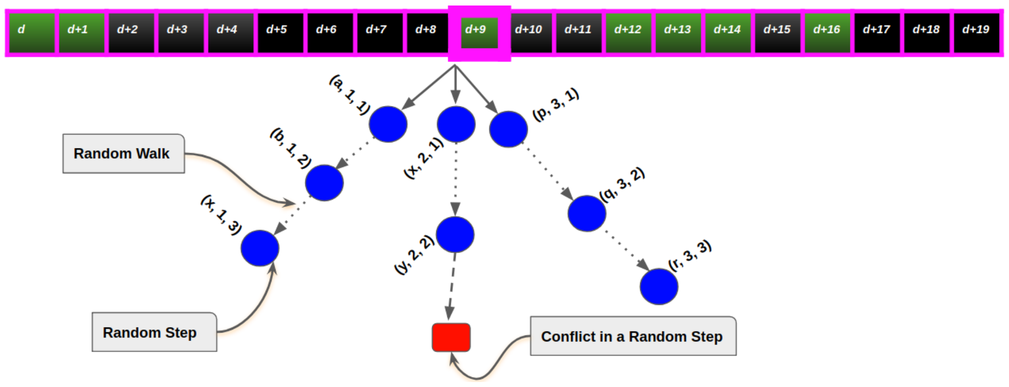
5.1 Exploration Guided CDCL Solving

Is it possible to correct the course of the search in a CD phase by identifying other promising variables that are currently under-ranked by `VSIDS`/`LRB`? In this work, we address this question by formulating the `expSAT` solver framework, which performs random explorations that probe into the future search space. The goal of this exploration is to discover branching variables that are likely to lead to conflicts from which clauses are learned.

Given a CDCL SAT solver, `expSAT` modifies it as follows:

- Before each branching decision, if a *substantial* CD phase is detected, then with probability p_{exp}, `expSAT` performs an *exploration episode*, which consists of a fixed number *nW* of random walks. Each walk consists of a limited number of *random steps*. Each such step consists of the uniform random selection of an unassigned *step variable*, followed by unit propagation (UP). A walk terminates either when a conflict occurs during UP, or after a fixed

Figure 5.1: The 20 adjacent cells denote 20 consecutive decisions starting from the d^{th} decision, with $d > 0$. A green cell denotes a decision with conflicts and a black cell denotes a decision without conflicts. After a CD phase $[d + 2 \ldots d + 8]$, just before taking decision $d+9$, expSAT performs an exploration episode via 3 random walks each limited to 3 steps. The second walk ends after 2 steps, due to a conflict. A triplet (v, i, j) represents that the variable v is randomly chosen at the j^{th} step of the i^{th} walk.



number *lW* of random steps have been taken. After each walk, the search state is restored and the next walk begins. Figure 5.1 illustrates an exploration episode with 3 walks and a maximum of 3 random steps per walk.

- An *exploration score* $exp(v)$ is computed for each step variable v.
- In the CDCL search, branching variables are chosen that maximize the expB heuristic. expB combines the exploration score of a variable and its activity score for the base heuristic B, where $B \in \{$VSIDS,LRB$\}$. Ties are broken randomly.
- All other elements, such as unit propagation, conflict analysis, restarts, and backjumping, remain the same as in the underlying CDCL SAT solver.

5.1.1 Algorithm Details

Input and Parameters

Given a SAT formula $\mathcal{F}$, let $uVars(\mathcal{F})$ be the set of currently unassigned variables in $\mathcal{F}$ and $assign(\mathcal{F})$ be the current partial assignment. The input to expSAT consists of $\mathcal{F}$ and four exploration parameters nW, lW, p_{exp}, ω, where $1 \leq nW$,

$lW \leq uVars(\mathcal{F})$, $0 < p_{exp}, \omega \leq 1$. All these parameters are explained above, except ω, which we explain below. When a random walk ends in a conflict after a series of random steps, some combination of the assigned variables has caused the conflict. In `expSAT`, we assign the most credit to the most recently assigned variable, and exponentially decay the credit for the variables assigned earlier in the walk, by a factor of ω per decision step. This approach is patterned on reward decay in reinforcement learning [76].

Algorithm 3: Exploration Based CDCL Solver: `expSAT`

```
Input: A CNF SAT formula: $\mathcal{F}$
       Exploration Parameters: $nW, lW, p_{exp}, \omega$
Output: Satisfiability of $\mathcal{F}$
1  Preprocess $\mathcal{F}$ and return result if it is solved;
2  while true do
3      $explore$ ←substantial_CD_Phase() and random() $\leq p_{exp}$;
4      if explore then
5          exploration_episode($nW, lW, p_{exp}, \omega$);
6      end
7      decide_and_assign_branching_variable();
8      while $true$ do
9          Perform Unit Propagation;
10         Break, if no new deductions are made;
11         Return SAT, if no more unassigned variables;
12         If a conflict is found, perform conflict analysis;
13            $cl$ ← obtain_learned_clause() ;
14            $bl$ ← obtain_backjumping_level($cl$) ;
15            if $bl$ is 0
16               Return UNSAT
17            Perform backtracking to $bl$;
18            Assign the asserting literal from the learned clause $cl$
19     end
20 end
```

Exploration in `expSAT`

Now we present the details of our `expSAT` algorithm. First, we define some notions used in the algorithm.

Definition 9: ***(Exploration Episode)*** *An exploration episode consists of nW walks,* *each containing a maximum of lW random steps. Each such step consists of two parts:*

- *Choose a step variable $v \in$ uVars($\mathcal{F}$) uniformly at random, and assign a boolean value to v using a polarity selection heuristic.*
- *After the assignment of v, run unit propagation. While performing unit propagation, any conflict ends the walk immediately.*

Definition 10: ***(Walk Scores)*** *Each variable v that participates in a walk during an exploration episode receives a walk score $ws(v)$, computed as:*

I. $ws(v) = 0$, if (i) the walk ended without a conflict, or (ii) the walk ended with a conflict and $lbd(c)$, the LBD score of the clause c derived from the current conflict, is greater than aLBD, the average LBD of learned clauses by the search (i.e., the quality of the derived clause c is below the search average). In case c is of lower quality than the search average, by assigning 0 to $ws(v)$, we prevent the prioritization of v, since this prioritization could result in learning a lower quality clause (such as c) during the search.

II. Otherwise, $ws(v) = \frac{\omega^d}{lbd(c)}$, with decay factor ω, and $d \geq 0$ the decision distance between variable v and the conflict which ended the current walk: If v was assigned at some step j during the current walk, and the conflict occurred after step $j' \geq j$, then $d = j' - j$. This score computation scheme provides more (resp. less) credit to the variable, which is closer (resp. further) to the current conflict.

Definition 11: ***(Exploration Scores)*** *The exploration score of a variable v, denoted exp(v), is the average of the walk scores, ws(v), of all random walks within the same episode in which v was one of the randomly chosen decision variables.*

The values of *ws(v)* and *exp(v)* for any variable v are always in the interval $[0, 1]$.

Example 2: Using the three random walks of Figure 5.1, we show how to compute ws and exp. Only the second walk produces a conflict. Let c be the derived clause from this conflict, with $lbd(c) = m <$ *aLBD*.

The walk and exploration scores for all variables participating in the first and third random walk are 0. As $lbd(c) < aLBD$, the variables x and y which participate in the second walk receive non-zero walk and exploration scores: $ws(y) = \frac{\omega^0}{m} = \frac{1}{m}$ and $ws(x) = \frac{\omega^1}{m}$. Since y only appears in this walk, but x appears in two walks, the exploration scores of y and x are, respectively, $\frac{1}{m}$ and $(\frac{\omega}{m})/2$.

The overhead of exploration must be balanced against its benefits. Hence, we perform exploration only amid substantial CD phases. We define a substantial CD phase as follows.

Definition 12: ***(Substantial CD Phases)*** *Let the ratio*

$$R = \frac{\#decisions\_without\_conflicts}{\#decisions\_with\_conflicts}$$

That is, $R+1$ *is the average number of decisions taken until one generates a conflict. A CD phase* $\langle c_s \rangle_j^k$ *is* substantial *if* $len(\langle c_s \rangle_j^k) \geq R$.

Algorithm 3 for `expSAT` algorithm is based on the standard CDCL algorithm. The changes are in lines 3-7. Line 3 checks whether an exploration episode should take place - it is triggered with probability p_{exp} within a substantial CD phase. Line 5 runs an exploration episode as described above. Line 7 selects a branching variable with maximum `expB` score, as described below.

Deciding the Branching Variable

Algorithm 4 shows the branching procedure for the `expSAT` framework. In `expSAT`, the branching variable is chosen by maximizing the `expB` score, that combines the exploration score and the activity score of an unassigned variable. To make both activity and exploration scores comparable, the raw exploration score is scaled by a factor f (Line 4, Algorithm 4), which depends on the currently active *base heuristic* $B \in \{$`LRB`,`VSIDS`$\}$.

Here, we present the `expSAT` extensions `expVSIDS` and `expLRB` of the base branching heuristics `VSIDS` and `LRB`. They use different values of f.

Algorithm 4: Decide the Branching Variable

```
Input: None
1 foreach $v \in uVars(\mathcal{F})$ do
2     exp($v$) ← compute_exp_score($v$);
3     $f$ ← scale_factor($B$) ;
4     expB($v$) ← activity($v$) + $f$ * exp($v$);
5 end
6 $v^* \leftarrow argmax_{v \in uVars(\mathcal{F})}$ expB($v$);
7 $assign(\mathcal{F}) \leftarrow assign(\mathcal{F}) \cup make\_assignment(v^*)$
```

The `expVSIDS` Branching Heuristic At any given search state, VSIDS increases the activity score of a given variable by using a bumping factor g^z, where $z > 0$ is the count of conflicts until that search state and $g > 1$ is a constant. This bumping factor is the same for all the variables at a given state of the search. With `VSIDS`, we use a scaling factor $f = g^z$ in line 3 of Algorithm 4.

The `expLRB` Branching Heuristic In contrast to `VSIDS`, the activity score increment method of `LRB` does not use an explicit bumping factor. Instead, it uses unique reward values for each variable derived from past conflict history. With `LRB`, we use $f = \max_{v \in uVars(\mathcal{F})} activity(v)$.

5.2 Experimental Evaluation of `expSAT`

5.2.1 Extension of Base Solver Systems with `expSAT`

We evaluate `expSAT` by extending the five baseline solvers gLCM, MplLRB, MplCOMSPS, MplCM, and MplCBT. While gLCM and MplLRB exclusively employ `VSIDS` and `LRB`, respectively, the other three systems use a combination of both.We summarize below how `VSIDS` and `LRB` are employed in these three systems:

- Both MplCOMSPS and MplCM employ `VSIDS` for the first 10,000 conflicts (Phase$^{\texttt{VSIDS1}}$), then switch to `LRB` for 2,500 seconds (Phase$^{\texttt{LRB}}$), and then switch back to `VSIDS` for the rest of the execution (Phase$^{\texttt{VSIDS2}}$).

- MplCBT employs `VSIDS` for the first 10,000 conflicts (Phase$^{\texttt{VSIDS1}}$), then switches to DIST for the next 40,000 conflicts (Phase$^{\texttt{DIST}}$), then switches to

LRB until 2,500 seconds have passed (Phase$^{\texttt{LRB}}$), and finally switches back to VSIDS for the rest of the execution (Phase$^{\texttt{VSIDS2}}$).

We implemented a total of 11 expVSIDS and expLRB extensions on top of these 5 baseline solvers[1]. We categorize these 11 solvers into three types:

- **expVSIDS Extensions (4 solvers):** We implemented expVSIDS for gLCM, MplCOMSPS, MplCM and MplCBT, which resulted in the solvers gLCM$^{\text{eV}}$, MplCOMSPS$^{\text{eV}}$, MplCM$^{\text{eV}}$, and MplCBT$^{\text{eV}}$, in which VSIDS is replaced by expVSIDS only in Phase$^{\texttt{VSIDS2}}$.

- **expLRB Extensions (4 solvers):** We implemented expLRB on top of MplLRB, MplCOMSPS, MplCM and MplCBT, which resulted in solvers MplLRB$^{\text{eL}}$, MplCOMSPS$^{\text{eL}}$, MplCM$^{\text{eL}}$, and MplCBT$^{\text{eL}}$. In these expSAT solvers, LRB is replaced by expLRB only in Phase$^{\texttt{LRB}}$.

- **Combined expLRB+expVSIDS Extensions (3 solvers):** We implemented both expVSIDS and expLRB in 3 systems MplCOMSPS, MplCM, and MplCBT, resulted in MplCOMSPS$^{\text{eLV}}$, MplCM$^{\text{eLV}}$, and MplCBT$^{\text{eLV}}$, where LRB is replaced by expLRB in Phase$^{\texttt{LRB}}$ and VSIDS is replaced by expVSIDS in Phase$^{\texttt{VSIDS2}}$.

5.2.2 Implementation Details

Three of our baseline solvers have Phase$^{\texttt{VSIDS1}}$, where VSIDS is run for a relatively short time (for the first 10,000 conflicts) for seeding these systems with clauses learned with VSIDS. To keep this setting unaltered, none of our extended solvers use expVSIDS in Phase$^{\texttt{VSIDS1}}$.

Our implementations are faithful to the expSAT algorithm described in the previous section, except for the following deviation that is induced by implementation features of the baseline solvers:

- For selecting a variable for making a decision, the base solvers use the following strategy: if the selected variable is currently unassigned, then it is selected.

[1]Source code for all of our expSAT systems are available at: https://github.com/solimul/expSAT_SourceCode.

Otherwise, the random step is skipped. For efficiency, the baseline solvers do not explicitly maintain the set $uVars(\mathcal{F})$. We have adopted the same strategy to implement random walks in all of our `expSAT` extensions. That is, for a given random step, if the randomly selected variable is already assigned, then we skip the selection, otherwise, we proceed with that random step.

For executing a random step while performing an exploration episode, all of our `expSAT` extensions use procedures such as, *unit-propagation* and *conflict-analysis*, which are already implemented in their corresponding baselines. In the `expSAT` solvers, these procedures are called by both search and exploration. This sharing may leave some side-effects of exploration in some data structures and may change the state of the search. We have instrumented our extensions to prevent most of the side-effects of performing exploration that we were able to detect. However, some of these side-effects were very complicated to remove.

A particular difficulty arose in the 2-watched literal scheme, which is used for efficient handling of clauses of a given formula[2]. Visitation in a clause while performing a random step may change the watched-literals in that clause. At the end of an exploration episode, we avoided restoring the watched literals (in the clauses visited during that exploration episode) back to the previous state because of implementation complexity and efficiency issues. It is important to note that this side effect does not affect the correctness of the `expSAT` solvers. At the end of a walk, we backtrack to the state where the walk started, and as a result, the clauses which were satisfied by the last walk becomes unsatisfied. The backtracking process assigns two literals in each of these unsatisfied clause as watched, although the watched literals in some of these unsatisfied clauses may get changed due to exploration. Since each of these unsatisfied clause has at least two watched literals after backtracking, the correctness of `expSAT` holds.

To set the values of the exploration parameters, we performed a small scale grid search with gLCM$^{\mathrm{eV}}$: we took one instance at random out of each of the 23 benchmark families in SAT-2018. We ran gLCM$^{\mathrm{eV}}$ on this subset of the 23 instances for

[2]In unit propagation, to avoid visiting all literals in clauses all time, a scheme called the 2-watched literal scheme has been proposed[49]

Table 5.1: Comparison between `expVSIDS` extensions and their baselines on $\mathbf{TS1}$. The PAR-2 scores (lower is better) are scaled down by the factor of $\frac{1}{10{,}000}$.

System	2017			2018			Combined	
	SAT	**UNSAT**	**SAT+UNSAT**	**SAT**	**UNSAT**	**SAT+UNSAT**	**SAT+UNSAT**	**PAR-2**
gLCM	82	**98**	**180**	95	97	192	372	4133
gLCM$^{\mathrm{eV}}$	**84**	95	179	**103**	97	**200**	**379 (+7)**	**4068**
MplCOMSPS	104	98	202	116	94	210	412	3701
MplCOMSPS$^{\mathrm{eV}}$	**106**	**101**	**207**	**125**	**96**	**221**	**428 (+16)**	**3442**
MplCM	103	111	214	128	100	228	442	3456
MplCM$^{\mathrm{eV}}$	**104**	111	**215**	128	100	228	**443 (+1)**	**3445**
MplCBT	97	110	207	133	102	235	442	3498
MplCBT$^{\mathrm{eV}}$	**98**	**113**	**211**	**138**	102	**240**	**451 (+9)**	**3400**

small parameter ranges, lW and nW in [4, 5, 6] and p_{exp} in [0.01, 0.02, 0.03]. From this grid search, we chose our default parameter setting of (mW, mS, p_{exp}) =(5, 5, 0.02). We set the value of the exponential decay parameter ω to 0.9 based on intuition. These values were used in all further experiments.

Next, we present three groups of experiments for each of these solvers on all instances from $\mathbf{TS1}$ and $\mathbf{TS2}$. We compare each baseline with its corresponding extension in terms of number of solved instances, PAR-2 score, and solution time.

5.2.3 Experiments with `expVSIDS` Extensions

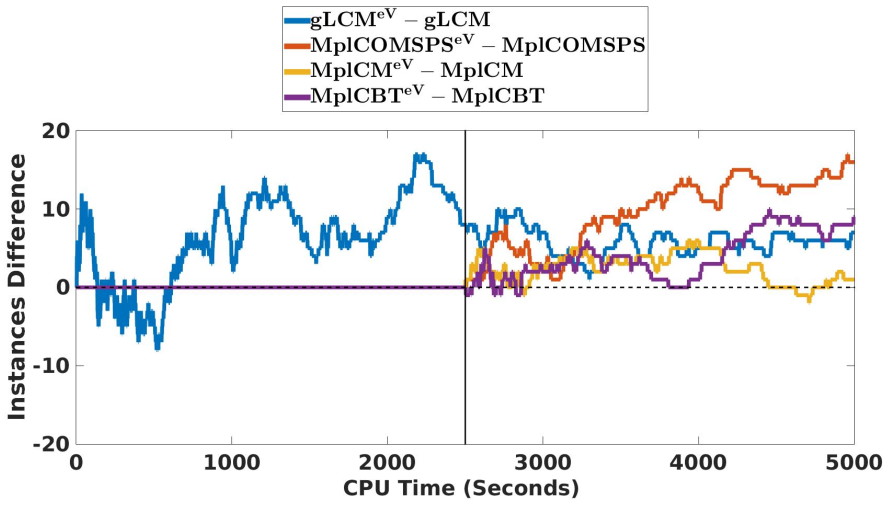
In our first set of experiments, we evaluate the `expVSIDS` heuristic by comparing the performance of the four baseline solvers gLCM, MplCOMSPS, MplCM, and MplCBT with their respective `expVSIDS` extensions gLCM$^{\mathrm{eV}}$, MplCOMSPS$^{\mathrm{eV}}$, MplCM$^{\mathrm{eV}}$, and MplCBT$^{\mathrm{eV}}$.

Comparison on TS1

Table 5.1 shows the results for $\mathbf{TS1}$ instances for the four `expVSIDS` extensions and their baseline solvers. Each `expVSIDS` extension solves more instances and has a lower PAR-2 score than its baseline.

For each of the Maple based systems, for a given instance, the runs with a baseline and its `expVSIDS` extension are identical prior to Phase$^{\mathrm{VSIDS2}}$. For these systems, only instances solved in Phase$^{\mathrm{VSIDS2}}$ show the impact of the `expVSIDS` approach.

Figure 5.2: Difference in solved instances over time for `expVSIDS` extensions and their baselines on **TS1**. The solid vertical line at 2500 seconds marks the start of phase$^{\texttt{VSIDS2}}$



The best performing `expVSIDS` extension, MplCOMSPS^{eV}, solves 16 more instances than its baseline. MplCBT^{eV} and MplCM^{eV} solve 9 and 1 more instances. gLCM^{eV} solves 7 more instances.

Figure 5.2 compares gLCM^{eV} (blue line), MplCOMSPS^{eV} (red line), MplCM^{eV} (yellow line) and MplCBT^{eV} (purple line) against their baselines in terms of number of instances solved as a function of time. For any time point above 0 on the vertical axis, our extensions solve more instances. The left side of the vertical solid line shows the instance difference that occurs before the starting of Phase$^{\texttt{VSIDS2}}$ and the right side shows the same measure that occurs at Phase$^{\texttt{VSIDS2}}$.

For the Maple based systems, `expVSIDS` is only used in Phase$^{\texttt{VSIDS2}}$. The solver MplCOMSPS^{eV} (red line) outperforms its baseline for all of this Phase. MplCBT^{eV} (purple line) and MplCM^{eV} (yellow line) outperform their baseline for most of this phase. For gLCM^{eV} (blue line), where `expVSIDS` is always active, the system performs slightly worse early on, but beats its baseline consistently from about 700 seconds.

Table 5.2: Results for `expVSIDS` extensions for **TS2**

System	SAT	UNSAT	Total
gLCM	3	4	7
gLCM^{eV}	**4**	**8**	**12 (+5)**
MplCOMSPS	2	2	4
MplCOMSPS^{eV}	**6**	**7**	**13 (+9)**
MplCM	0	1	1
MplCM^{eV}	**6**	**4**	**10 (+9)**
MplCBT	21	20	41
MplCBT^{eV}	**23**	20	**43 (+2)**

Comparison on TS2

SAT-2018 includes 17 SATCoin instances. In the experimental results reported in Table 5.1, compared to their baselines, MplCM^{eV} and MplCBT^{eV} solve an equal number of instances over these 17 instances. However, over this subset of SAT-2018 instances, we observe strong performance gains with `expVSIDS` extensions gLCM^{eV} (+5) and MplCOMSPS^{eV} (+6). We further evaluate `expSAT` on a SATCoin benchmark by generating 52 hard instances (**TS2**), which differ from the SAT-2018 instances.

Table 5.2 evaluates `expVSIDS` extensions for **TS2**. The best performing `expVSIDS` extensions, MplCM^{eV} and MplCOMSPS^{eV}, solve 10 and 13 instances respectively, beating their baselines by 9. gLCM^{eV} and MplCBT^{eV} solve 5 and 2 additional instances. Figure 5.3 shows the difference in solved instances for these 8 solvers on **TS2**. All of our `expVSIDS` extensions solve these problems more quickly at most time points.

5.2.4 Experiments with `expLRB` Extensions

This set of experiments evaluates four `expLRB` extensions MplLRB^{eL}, MplCOMSPS^{eL}, MplCM^{eL}, and MplCBT^{eL} against their respective baselines MplLRB, MplCOMSPS, MplCM, and MplCBT for both **TS1** and **TS2**.

Figure 5.3: Difference in solved instances over time for `expVSIDS` extensions and their baselines on **TS2**

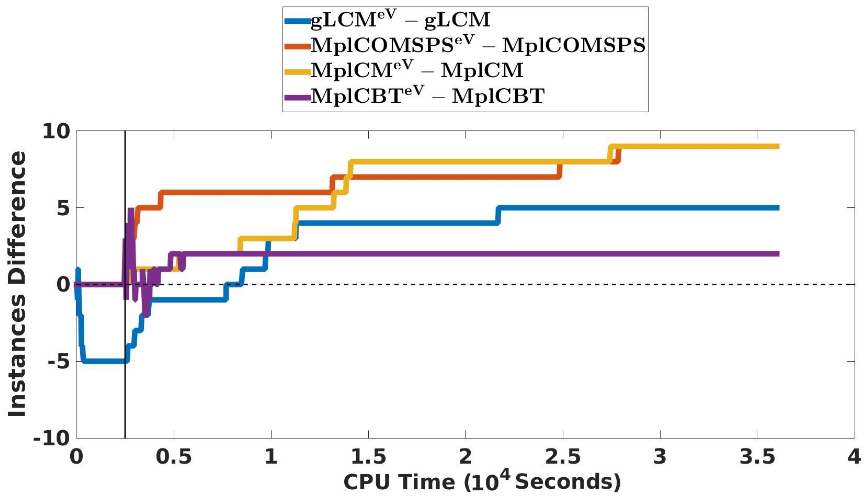


Table 5.3: Comparison between `expLRB` extensions and their baselines on **TS1**

System	2017			2018			Combined	
	SAT	**UNSAT**	**SAT+UNSAT**	**SAT**	**UNSAT**	**SAT+UNSAT**	**SAT+UNSAT**	**PAR-2**
MplLRB	79	**95**	**174**	111	93	204	378	4028
MplLRB^{eL}	**84**	89	173	**129**	91	**220**	**393 (+15)**	**3879**
MplCOMSPS	104	98	202	116	94	210	412	3701
MplCOMSPS^{eL}	103	**100**	**203**	**124**	**96**	**220**	**423 (+11)**	**3615**
MplCM	103	111	214	**128**	**100**	**228**	**442**	**3456**
MplCM^{eL}	103	111	214	125	99	224	438 (-4)	3482
MplCBT	97	110	207	133	102	235	442	3498
MplCBT^{eL}	**101**	110	**211**	**138**	100	**238**	**449 (+7)**	**3414**

Results for TS1

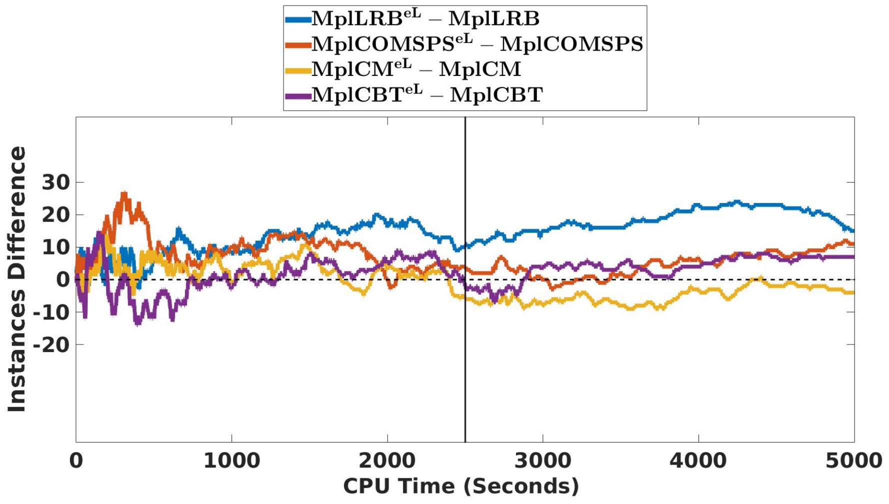
Table 5.3 evaluates our four `expLRB` extensions on **TS1**. Three solvers MplLRB^{eL} (+15), MplCOMSPS^{eL}(+11), and MplCBT^{eL} (+7) show strong gains, solving many additional instances, and achieving significantly lower PAR-2 scores. MplCM^{eL} solves 4 fewer instances compared to its baseline. However, despite being penalized heavily for the 4 unsolved instances, it solves quite a few instances faster than its baseline and the PAR-2 scores are fairly close.

Figure 5.4 compares the solving speed of the `expLRB` extensions with their baselines. The three `expLRB` extensions that achieve overall improvement, solve more problems at most time points, except for the first 1,000 seconds for MplCBT^{eL}

(purple line in Figure 5.4).

MplCM^{eL} (yellow line, Figure 5.4) performs better than its baseline before 2,500 seconds, while `expLRB` is active before the start of Phase$^{\texttt{VSIDS2}}$. However, it slows down after 2,500 seconds in Phase$^{\texttt{VSIDS2}}$. Hence, when active, `expLRB` does not have a bad effect on MplCM^{eL}, however, the combination of `expLRB` and `VSIDS` does not work well for MplCM^{eL}.

Figure 5.4: Difference in solved instances over time for `expLRB` extensions and their baselines on **TS1**



Results for TS2

Table 5.4 compares our `expLRB` extensions with their respective baselines. Remarkably, MplLRB^{eL}, in which `expLRB` is active over the whole run of the solver (timeout is 36,000), solves 46 out of 52 hard SATCoin instances, while the baseline solves none. Thus for this test set and this solver, `expLRB` scales extremely well.

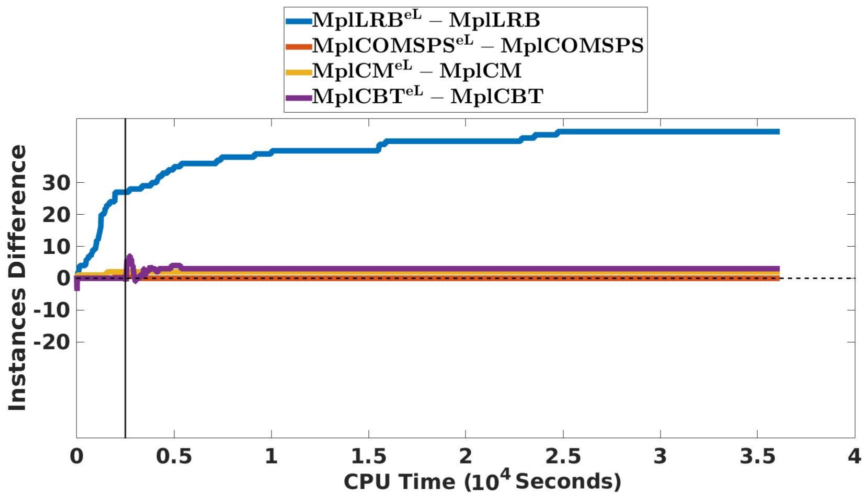
Both MplCM^{eL} and MplCBT^{eL} solve more problems than their baselines. The extension MplCOMSPS^{eL} solves an equal number of problems as its baseline.

Figure 5.5 shows that for **TS2**, MplLRB^{eL} solves many instances quickly, solving 40 instances within 5,000 seconds, and 6 more instances before the timeout of 36,000 seconds. The minor performance change for the other three systems is also shown in Figure 5.5.

Table 5.4: Results for `expLRB` extensions for **TS2**

System	**SAT**	**UNSAT**	**Total**
MplLRB	0	0	0
MplLRB$^{\mathrm{eL}}$	**24**	**22**	**46 (+46)**
MplCOMSPS	2	2	4
MplCOMSPS$^{\mathrm{eL}}$	1	**3**	4
MplCM	0	1	1
MplCM$^{\mathrm{eL}}$	**2**	**1**	**3 (+2)**
MplCBT	21	20	41
MplCBT$^{\mathrm{eL}}$	**22**	**22**	**43 (+2)**

Figure 5.5: Difference in solved instances over time for `expLRB` extensions and their baseline on **TS2**



Experiments with Extended Phase$^{\mathrm{LRB}}$ In MplCOMSPS$^{\mathrm{eL}}$, MplCM$^{\mathrm{eL}}$ and MplCBT$^{\mathrm{eL}}$, `expLRB` is active only in Phase$^{\mathrm{LRB}}$ (first 2,500 seconds of their execution) and switches to `VSIDS` until timeout (36,000 seconds for **TS2**). In contrast, `expLRB` remains active for whole run of MplLRB$^{\mathrm{eL}}$, the best performing `expLRB` solver for this test set. The modest performance gains with the other three `expLRB` solvers for **TS2** could be due to the shorter period of activation of `expLRB` in these three solvers.

To test this hypothesis, we repeated the same experiment for **TS2** instances with a slightly modified version of MplCOMSPS$^{\mathrm{eL}}$, the weakest of the four `expLRB`

Table 5.5: Comparison between `expLRB+expVSIDS` extensions and their baselines on **TS1**

System	2017			2018			Combined	
	SAT	**UNSAT**	**SAT+UNSAT**	**SAT**	**UNSAT**	**SAT+UNSAT**	**SAT+UNSAT**	**PAR-2**
MplCOMSPS	104	98	202	116	94	210	412	3701
MplCOMSPS^{eLV}	**105**	97	202	**127**	**96**	**223**	**425 (+13)**	**3601**
MplCM	103	**111**	214	**128**	**100**	**228**	**442**	**3456**
MplCM^{eLV}	**105**	109	214	123	99	222	436 (-6)	3492
MplCBT	97	110	207	133	102	235	442	3498
MplCBT^{eLV}	**102**	110	**212**	**138**	101	**239**	**451 (+9)**	**3403**

solvers for **TS2** (Table 5.4). In this experiment, Phase$^{\texttt{LRB}}$ in MplCOMSPS^{eL} is extended to 18,000 seconds from the default 2,500 seconds. This version solves equal number of instances (4) as the default MplCOMSPS^{eL}. Thus the extension of Phase$^{\texttt{LRB}}$ in this solver does not lead to solving of more instances.

5.2.5 Experiments with both `expLRB` and `expVSIDS` Extensions

Our last set of experiments evaluates MplCOMSPS^{eLV}, MplCM^{eLV}, and MplCBT^{eLV}, in which both `LRB` and `VSIDS` are replaced by `expLRB` and `expVSIDS` for Phase$^{\texttt{LRB}}$ and Phase$^{\texttt{VSIDS2}}$.

Results for TS1

Table 5.5 and Figure 5.6 compares three `expLRB+expVSIDS` extensions with their respective baselines. MplCOMSPS^{eLV} (+13) and MplCBT^{eLV} (+9) solve significantly more instances. Both systems also achieve a lower PAR-2 score than their baseline. MplCM^{eLV} performs poorly (-6). This result is similar to MplCM^{eL} (-4).

Results for TS2

Evaluation of `expLRB+expVSIDS` solvers for **TS2** are shown in Table 5.6. The performance gains are similar to the `expLRB` and `expVSIDS` extensions. Figure 5.7 shows that each of these three extensions solves instances faster than their baselines.

Figure 5.6: Difference in solved instances over time for `expLRB+expVSIDS` extensions and their baselines on **TS1**

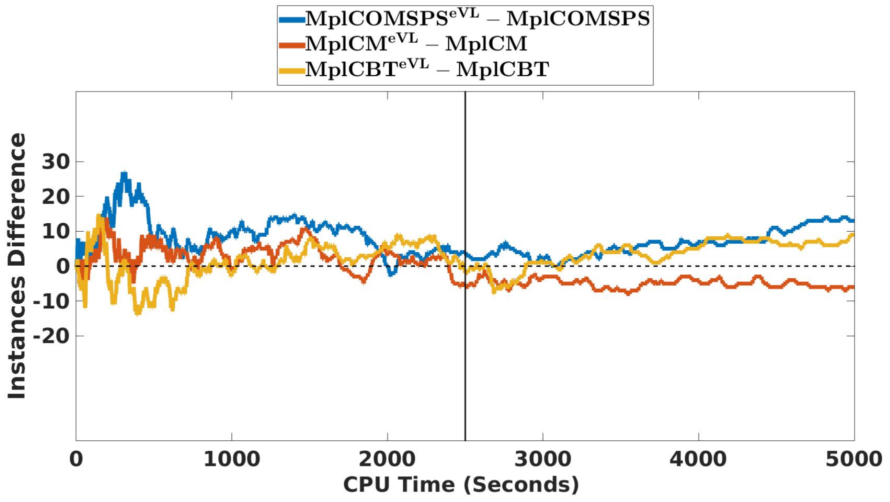


Table 5.6: Results for `expLRB+expVSIDS` extensions for **TS2**

System	**SAT**	**UNSAT**	**Total**
MplCOMSPS	2	2	4
MplCOMSPS^{eLV}	**6**	**7**	**13(+9)**
MplCM	0	1	1
MplCM^{eLV}	**5**	**5**	**10 (+9)**
MplCBT	21	20	41
MplCBT^{eLV}	**23**	**21**	**44 (+3)**

5.3 Detailed Analysis of the Experimental Results

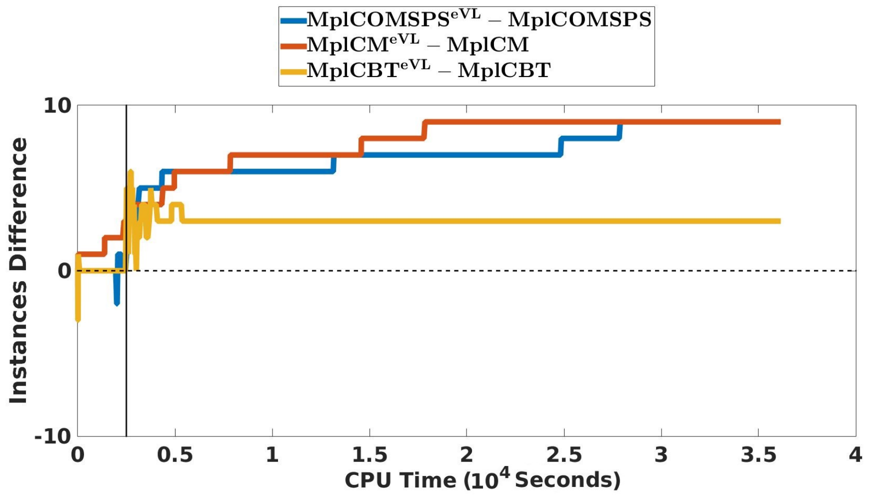
For a detailed analysis of the results presented in the previous section, we gather experimental data from

- gLCM and gLCM^{eV} to analyze the performance gain for `expVSIDS`.
- MplLRB and MplLRB^{eL} to analyze the performance gain for `expLRB`.

For convenience of the analysis, for a given set of instances, we define four subsets below.

- **exp**$^{g+}$: Instances which are solved by gLCM^{eV} but not by gLCM, or for which gLCM takes longer time to solve than gLCM^{eV}.

Figure 5.7: Difference in solved instances over time for `expLRB+expVSIDS` extensions and their baselines on **TS2**



- **exp**$^{g-}$: Instances which are solved by gLCM but not by gLCM^{eV}, or where gLCM^{eV} takes longer.

- **exp**$^{m+}$: Instances solved by MplLRB^{eL} but not by MplLRB, or for which MplLRB takes longer time to solve than MplLRB^{eL}.

- **exp**$^{m-}$: Instances solved by MplLRB but not by MplLRB^{eL}, or where MplLRB^{eL} takes longer.

5.3.1 GLR and Average LBD Score

In [41], it is shown that on average, a more efficient CDCL branching heuristic leads to higher GLR values and lower average LBD (aLBD) scores of the learned clauses. Here we analyze these scores on our **exp** test set.

Comparison between `VSIDS` and `expVSIDS`

The top two rows of Table 5.7 show the average GLR and average aLBD values for **exp**$^{g-}$ and **exp**$^{g+}$ for **TS1** with gLCM and gLCM^{eV}.

- For **exp**$^{g-}$, where the baseline gLCM is more efficient, gLCM has a lower average GLR score. However, gLCM learns higher quality clauses (lower

Table 5.7: Comparison of average GLR and average aLBD

Instance Set	System	$\mathbf{exp}^{g-}$			$\mathbf{exp}^{g+}$		
		#inst	avg. GLR	avg. aLBD	#	avg. GLR	avg. aLBD
TS1 (2017 + 2018)	gLCM	156	0.47	**15.08**	245	0.49	15.88
	eGLCM		**0.49**	15.79		0.49	**14.78**
TS2 (SATCoin)	gLCM	6	**0.61**	**25.46**	11	0.34	35.81
	eGLCM		0.30	33.55		**0.37**	**32.29**

aLBD), on average.

- For $\mathbf{exp}^{g+}$, where gLCM$^{\text{eV}}$ is more efficient, gLCM$^{\text{eV}}$ generates conflicts at the same rate as the baseline. However, it learns higher quality clauses (lower aLBD), on average.

Hence, our results on **TS1** for `VSIDS` and `expVSIDS` are largely consistent with the observation of [41].

Table 5.8: Comparison of average GLR and average aLBD in MplLRB and MplLRB$^{\text{eL}}$

Instance Set	System	$\mathbf{exp}^{m-}$			$\mathbf{exp}^{m+}$		
		#inst	avg. GLR	avg. aLBD	#	avg. GLR	avg. aLBD
TS1 (2017+ 2018)	MplLRB	225	0.46	**16.08**	194	**0.47**	19.92
	MplLRB$^{\text{eL}}$		**0.48**	16.80		0.46	**17.64**
TS2 (SATCoin)	MplLRB	0	–	–	46	0.02	51.41
	MplLRB$^{\text{eL}}$		–	–		**0.03**	**40.11**

The experimental data from **TS2** for `VSIDS` and `expVSIDS` are consistent with [41]. The bottom two rows of Table 5.7 show that in each case, the better system has both a higher average GLR and a lower average aLBD.

Comparison between `LRB` and `expLRB`

The top two rows of Table 5.8 show the average GLR and average aLBD values for $\mathbf{exp}^{m-}$ and $\mathbf{exp}^{m+}$ for **TS1** with MplLRB and MplLRB$^{\text{eL}}$, respectively. For these two subsets of instances of **TS1**, the better system achieves slightly lower GLR values on average. However, the better system achieves a lower average aLBD, which is consistent with the observation of [41]. This is particularly pronounced for the case of $\mathbf{exp}^{m+}$ for both **TS1** and **TS2** in Table 5.8, where the better system

Table 5.9: Comparison of average CD phase length with gLCM and gLCM$^{\mathrm{eV}}$

Instance Set	System	$\mathbf{exp}^{g-}$		$\mathbf{exp}^{g+}$	
		#inst	avg. CDLen	#	avg. CDLen
TS1 (2017+ 2018)	gLCM	156	30.27	245	8.60
	gLCM$^{\mathrm{eV}}$		**29.97**		**8.26**
TS2 (SATCoin)	gLCM	6	**4.96**	11	8.27
	gLCM$^{\mathrm{eV}}$		9.38		**7.73**

MplLRB$^{\mathrm{eL}}$ achieves lower average aLBD (average aLBD reduction of 2.28=19.92-17.64 for **TS1** and 11.30=51.41-40.11 for **TS2**)- a significant improvement in quality of the learned clauses.

For the 46 instances from **TS2**, which are exclusively solved by MplLRB$^{\mathrm{eL}}$ (bottom two rows of Column $\mathbf{exp}^{m+}$ of Table 5.8), the system achieves a higher average GLR and a lower average aLBD. This case is strongly consistent with [41]. The baseline MplLRB could not solve any of these SATCoin instances, hence no data is available for the bottom two rows of Column $\mathbf{exp}^{m-}$ of Table 5.8.

5.3.2 Reduction of Average CD Phase Length

Table 5.9 shows that for both **TS1** and **TS2**, exploration in gLCM$^{\mathrm{eV}}$ slightly reduces the average CD phase length for both $\mathbf{exp}^{g-}$ and $\mathbf{exp}^{g+}$ in three out of four cases.

Table 5.10 shows that MplLRB$^{\mathrm{eL}}$ has a lower average CD phase than MplLRB for both **TS1** and **TS2** in all three cases for which data are available.

Overall, our analysis clearly shows that random exploration helps solvers to escape from CD phases more swiftly.

Table 5.10: Comparison of average CD phase length with MplLRB and MplLRB$^{\mathrm{eL}}$

Instance Set	System	$\mathbf{exp}^{m-}$		$\mathbf{exp}^{m+}$	
		#inst	avg. CDLen	#	avg. CDLen
TS1 (2017+ 2018)	MplLRB	225	96.89	194	18.19
	MplLRB$^{\mathrm{eL}}$		**95.05**		**17.18**
TS2 (SATCoin)	MplLRB	0	–	46	98.63
	MplLRB$^{\mathrm{eL}}$		–		**88.04**

5.4 Exploration Parameter Adaptation

A parameter setting that is effective for one instance may not be effective for another. We developed an algorithm named *ParamAdapt* to dynamically control when to trigger exploration episodes, and how much exploration to perform in each episode.

5.4.1 The ParamAdapt Algorithm

The three exploration parameters nW, lW, and p_{exp} are adapted between restarts based on the search behavior. A parameter setting is a triple $\Sigma = (nW, lW, p_{exp})$, which is updated at the beginning of each restart by *ParamAdapt* by comparing the exploration performance of the two most recent search periods: the period between the latest two restarts and the period before it. The search in `expSAT` starts with a default value of Σ, $\Sigma' = (nW', lW', p'_{exp})$. *ParamAdapt* keeps track of the following statistics about all exploration steps within a period: the number of random steps $rSteps$, the number of conflicts c, the number of glue-clauses gc, and the mean LBD value lbd of the learned clauses.

With fixed weights $w_1 > w_2 > w_3$, an *exploration performance metric* (EPM) σ is defined as

$$\sigma = \frac{w_1 \times gc + w_2 \times c}{rSteps} + w_3 * \frac{1}{lbd}$$

This performance metric rewards finding glue clauses (most important), finding any conflict (very important), and learning clauses with low LBD score (important).

At each restart, the algorithm computes a new EPM σ^{new} and compares (the comparison starts after the second restart) it with the prior one σ^{old}, and computes a new parameter settings Σ^{new} from Σ^{old}.

- If $\sigma^{new} < \sigma^{old}$, the performance of exploration is worse than before. First, Σ^{new} is modified from Σ^{old} by performing an *increment*: Randomly select a parameter $p \in \Sigma^{old}$ and increase its value by a predefined *stepsize*.

- If $\sigma^{new} = \sigma^{old}$, we perform an *increment*.

- If $\sigma^{new} > \sigma^{old}$, then exploration is working better than before. We do not change Σ^{old} in this case.

The values of a parameter are bounded by a range. Whenever a value leaves its range, it is reset to its default value.

Table 5.11: Parameter values for the adaptive-`expSAT` solvers

Description	**Parameters**	**Value**
Weights	(w_1, w_2, w_3)	$(40, 10, 3)$
Range for nW	$[l_{nW}, u_{nW}]$	$[1, 20]$
Range for lW	$[l_{lW}, u_{lW}]$	$[1, 10]$
Range for p_{exp}	$[l_{p_{exp}}, u_{p_{exp}}]$	$[0.02, 0.6]$
Step size for parameters	$(s_{nW}, s_{lW}, s_{p_{exp}})$	$(1, 1, 0.01)$
Exponential Decay Factor	ω	0.9

5.4.2 Experiments

To test the potential of *ParamAdapt*, we perform experiments for $\mathbf{TS1}$ and $\mathbf{TS2}$ with four `expVSIDS` extensions with *ParamAdapt* implemented on each of them. We denote by $\Psi^{\text{eV\_ad}}$ the adaptive version of non-adaptive `expVSIDS` extension Ψ^{eV}.

We set the default parameters $\Sigma' = (nW', lW', p'_{exp}) = (5, 5, 0.02)$. The values of the other parameters are given in Table 5.11.

Table 5.12: Comparison between adaptive `expVSIDS` extensions and non-adaptive `expSAT` extensions with $\mathbf{TS1}$

System	**2017**			**2018**			**Combined**	
	SAT	**UNSAT**	**SAT+UNSAT**	**SAT**	**UNSAT**	**SAT+UNSAT**	**SAT+UNSAT**	**PAR**
gLCM$^{\text{eV}}$	84	**95**	179	103	**97**	200	379	4068
gLCM$^{\text{eV\_ad}}$	**85**	94	179	**105**	96	**201**	**380 (+1)**	**4036**
MplCOMSPS$^{\text{eV}}$	**106**	**101**	**207**	**125**	96	**221**	**428 (+11)**	**3442**
MplCOMSPS$^{\text{eV\_ad}}$	99	99	198	123	96	219	417	3665
MplCM$^{\text{eV}}$	104	**111**	215	128	100	228	443	3445
MplCM$^{\text{eV\_ad}}$	**107**	108	215	**130**	100	230	**445 (+2)**	**3435**
MplCBT$^{\text{eV}}$	98	**113**	**211**	**138**	102	**240**	**451 (+4)**	**3400**
MplCBT$^{\text{eV\_ad}}$	**99**	111	210	135	102	237	447	3410

Table 5.12 shows the performance comparison between the non-adaptive and adaptive `expVSIDS` extensions. For $\mathbf{TS1}$, the overall performance of the non-adaptive versions is better: MplCOMSPS$^{\text{eV}}$ and MplCBT$^{\text{eV}}$ solve 11 and 4 more

problems compared to their adaptive versions. gLCM$^{eV\_ad}$ solves 1 more instance than gLCM^{eV}. MplCM$^{eV\_ad}$ solves 2 more instances compared to its non-adaptive version.

For **TS2**, the performance of most of the adaptive `expVSIDS` solvers is significantly better than their respective non-adaptive versions, as shown in Table 5.13. MplCOMSPS$^{eV\_ad}$ and MplCM$^{eV\_ad}$ solve 16 and 13 more problems than MplCOMSPS^{eV} and MplCM^{eV}, respectively. gLCM$^{eV\_ad}$ solves 9 more instances than gLCM^{eV}. MplCBT$^{eV\_ad}$ and MplCBT^{eV} solve an equal number of problems.

Table 5.13: Additional comparison with **TS2**

System	**SAT**	**UNSAT**	**Total**
gLCM^{eV}	4	8	12
gLCM$^{eV\_ad}$	**13**	8	**21 (+9)**
MplCOMSPS^{eV}	6	7	13
MplCOMSPS$^{eV\_ad}$	**14**	**15**	**29 (+16)**
MplCM^{eV}	6	4	10
MplCM$^{eV\_ad}$	**14**	**9**	**23 (+13)**
MplCBT^{eV}	**23**	20	43
MplCBT$^{eV\_ad}$	22	**21**	43

5.4.3 Analysis of Experimental Results with ParamAdapt

Table 5.14: Comparison of performance metric between MplCOMSPS^{eLV} and MplCOMSPS$^{eV\_ad}$

1: System	**2: Overhead**	**3: exp_GLR**	**4: exp_aLBD**	**5: GLR**	**6: aLBD**	**7: CDLen**
MplCOMSPS^{eV}	**48.51 secs**	**0.0193**	**15.25**	0.51	**22.91**	**20.08**
MplCOMSPS$^{eV\_ad}$	198.21 secs	0.0179	15.78	0.51	23.39	20.77

For **TS1**, the performance of the adaptive `expVSIDS` extensions is worse. Here, we present an analysis that explains this result.

Table 5.14 shows an analysis with MplCOMSPS^{eV} and MplCOMSPS$^{eV\_ad}$, for which we observed the largest performance gap for **TS1**. MplCOMSPS^{eV} has significantly lower overhead incurred in exploration (Column 2), exploration finds conflicts at a faster rate (Column 3), from which lower LBD (Column 4) clauses are derived. During the search both systems generate clauses at the same rate (Column 5), however the average LBD of the learned clauses for MplCOMSPS^{eV} is lower (Column 6), so is the average CD phase length for MplCOMSPS^{eV} (Column 7). These data may just explain the better performance of MplCOMSPS^{eV} over MplCOMSPS$^{eV\_ad}$ for **TS1**.

5.5 Discussion of Experimental Results

5.5.1 Comparing the Relative Strength and Weakness of `expVSIDS` and `expLRB`

In Table 5.15, we compare the performance of our `expVSIDS` and `expLRB` extensions for **TS1** instances.

- For the common problems solved by both `expVSIDS` and `expLRB` extensions, `expLRB` extensions are usually faster, as shown in Column 2.
- However, when counting instances which are solved by only one system, `expVSIDS` extensions usually do better, as shown in Column 3. This results in more problems solved for the `expVSIDS` extensions (Column 4).

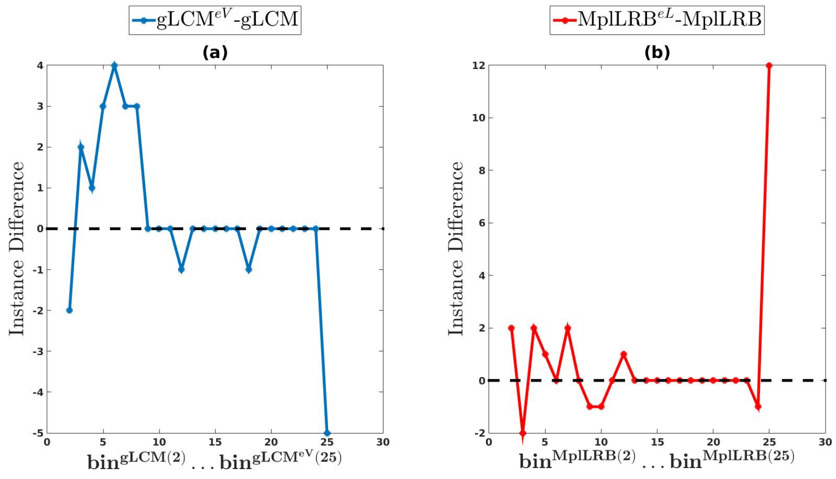
Table 5.15: Comparison between `expVSIDS` and `expLRB` extensions

1:Systems	**2:Common Solved**		**3:Exclusively Solved**		**4:Total**	
	Count	**avg. Solve Time**	**Count**	**avg. Solve Time**	**Count**	**avg. Solve Time**
gLCM^{eV}	325	827.44	54	1656.9	379	945.63
MplLRB^{eL}	325	**612.8**	**68**	1621.8	**393**	787.36
MplCOMSPS^{eV}	400	788.58	**28**	2329.2	**428**	889.37
MplCOMSPS^{eL}	400	**751.78**	23	1942.8	423	816.54
MplCM^{eV}	422	794.01	**21**	1962.7	**443**	849.41
MplCM^{eL}	422	**783.74**	16	1978.0	438	827.36
MplCBT^{eV}	423	**824.05**	**28**	2389.9	**451**	921.27
MplCBT^{eL}	423	828.26	26	2045.2	449	898.72

5.5.2 Which CD Phase Bins Correspond to Gains?

In Figures 4.2 and 4.4 (Chapter 4), for **TS1**, we have observed that the distribution of instances based on average CD phase length is heavy-tailed (i.e, there are many instances with long CD phase, on average) for both `VSIDS` and `LRB`, respectively.

Figure 5.8: Difference in solved instances between two `expSAT` extensions and their baselines over bins of instances.



Does the performance gain achieved with the `expSAT` approach come from these instances with high average CD phase lengths? To answer this question, we compare (a) gLCM and gLCM$^{\mathrm{eV}}$, and (b) MplLRB and MplLRB$^{\mathrm{eL}}$.

For each of the 25 bins b in the right plots of Figure 4.2 (section 4),

- let $\mathbf{bin}^{\mathbf{gLCM(b)}} \subset \mathbf{TS1}$ be the set of instances in the b^{th} leftmost bins in Figure 4.2.

Similarly, for each of the 25 bins b in the right plots of Figure 4.4 (Chapter 4),

- let $\mathbf{bin}^{\mathbf{MplLRB(b)}} \subset \mathbf{TS1}$ be the set of instances in the b^{th} leftmost bins in Figure 4.4.

In Figure 5.8, the left plot compares the difference of the number of solved instances by gLCM$^{\mathrm{eV}}$ and gLCM for the instances in $\mathbf{bin}^{\mathbf{gLCM(2)}}, \ldots, \mathbf{bin}^{\mathbf{gLCM(25)}}$. For gLCM$^{\mathrm{eV}}$, the gains come with instances from $\mathbf{bin}^{\mathbf{gLCM(3)}}, \ldots, \mathbf{bin}^{\mathbf{gLCM(7)}}$. Hence the gains in case of gLCM$^{\mathrm{eV}}$ come from those instances for which the baseline has moderately high average CD phase length.

The right plot in Figure 5.8 compares the difference of the number of solved instances by MplLRB$^{\mathrm{eL}}$ and MplLRB for the instances in $\mathbf{bin}^{\mathbf{MplLRB(2)}}, \ldots,$ $\mathbf{bin}^{\mathbf{MplLRB(25)}}$. For MplLRB$^{\mathrm{eL}}$, some gains come from bins (in between $\mathbf{bin}^{\mathbf{MplLRB(2)}}$,

$\ldots$, $\mathbf{bin}^{\mathbf{gLCM(12)}}$) for which its baseline has low-to-moderately long CD phase, on average, and the biggest gains come from $\mathbf{bin}^{\mathbf{gLCM(25)}}$, for which, on average, the baseline has very long CD phase.

This analysis indicates that the gains with the `expSAT` approach can come from instances irrespective of their average CD phase length and is more dependant on solvers that implement the approach and the benchmark instances. In our analysis, $\mathbf{bin}^{\mathbf{MplLRB(25)}}$ corresponds to the biggest gains with MplLRB. Out of these 12 instances from $\mathbf{bin}^{\mathbf{MplLRB(25)}}$, 8 instances were also solved by gLCM$^{\mathrm{eV}}$ in the following bins: $\mathbf{bin}^{\mathbf{gLCM(6)}}$, $\mathbf{bin}^{\mathbf{gLCM(5)}}$, and $\mathbf{bin}^{\mathbf{gLCM(4)}}$. On average, these 8 instances, which are solved by both MplLRB$^{\mathrm{eL}}$ and gLCM$^{\mathrm{eV}}$, have very long CD phases with MplLRB, but have moderately long CD phases with gLCM.

5.5.3 Overhead and Conflict Discovery Rate of Exploration

What is the cost and benefit of performing exploration? On average, for the instances from $\mathbf{TS1}$, gLCM$^{\mathrm{eV}}$ (resp. MplLRB$^{\mathrm{eL}}$) incurs 92.53 (resp. 76.51) seconds of overhead to perform exploration, which is about 3.94% (resp. 2.72%) of its average run time of 2346.12 (resp. 2791.31 secs) seconds. With random exploration amid substantial CD phases, on average, gLCM$^{\mathrm{eV}}$ (resp. MplLRB$^{\mathrm{eL}}$) finds about 2 (resp. 3.4) conflicts per 100 random steps.

5.5.4 Performing Exploration More Frequently Just After Restarts

In Section 4.8, we have shown that on average, BFC (Before First Conflict) CD phases are significantly longer than AFC (After First Conflict) phases for almost all of the $\mathbf{TS1}$ instances. Typically, restarts are followed by longer CD phases in the upper part of a search tree. Does performing more exploration amid the BFC CD phases help?

Experiment

We evaluated MplLRB$^{\mathrm{eL}}{}_{\mathrm{res}}$, a version of MplLRB$^{\mathrm{eL}}$ where we perform exploration before every decision amid a BFC CD phase, until it ends with the discovery of a conflict. That is MplLRB$^{\mathrm{eL}}{}_{\mathrm{res}}$ performs exploration more aggressively amid BFC

CD phases and turns off exploration amid AFC CD phases. This version solves 384 instances out of 750 **TS1** instances. In comparison, regular MplLRB^{eL} solves 393 instances and the baseline MplLRB solves 378 instances. Hence, performing exploration aggressively only amid BFC CD phases is helpful, but not as good as performing exploration amid all substantial CD phases through out the course of the search.

Analysis of Overhead of Performing Exploration

On average, MplLRB$^{eL}{}_{res}$ incurs a significantly higher exploration overhead of 279.94 seconds in comparison to MplLRB^{eL}, with 76.51 seconds of exploration overhead. Furthermore, with the higher exploration overhead, exploration in MplLRB$^{eL}{}_{res}$ discovers conflicts at a slower rate (2.98 per 100 random steps) than MplLRB^{eL} (3.4 per 100 random steps). Hence, compared to MplLRB^{eL}, with more overhead, exploration in MplLRB$^{eL}{}_{res}$ discovers conflicts at a lower rate. These two observations may just explain the worse performance of MplLRB$^{eL}{}_{res}$.

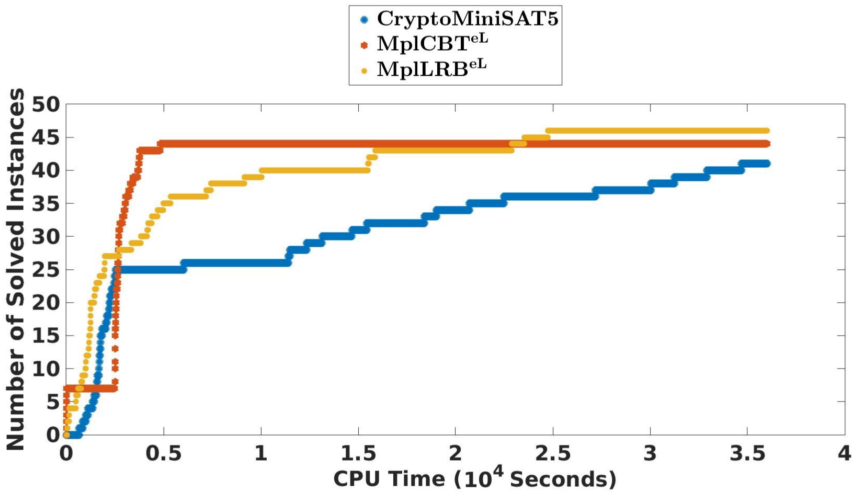
5.5.5 Learning Derived Clauses from Exploration

In `expSAT`, we do not learn the clauses that are derived from conflicts discovered during exploration. This is based on the following intuition. Whenever a CDCL search learns a clause c, c is immediately used by propagating the asserting literal (first UIP), which is hosted by c. However, in case of exploration, when a clause c is derived, such propagations do not immediately follow. Thus the utility of c is uncertain. We ran an experiment with gLCM^{eV} for instances from SAT-2017, where we saved the learned clauses that are derived during exploration. This version of gLCM^{eV} (solves 178) is than the regular gLCM^{eV} (solves 179) reported in Table 5.1.

5.5.6 CryptoMiniSAT on TS2

On **TS1**, almost all of our `expSAT` extended solvers show strong performance gains over its baseline. To put this experiment into perspective, we ran experiment with

Figure 5.9: Number of Solved Instances as function of of time for CryptoMiniSAT5, MplCBT^{eLV} and MplLRB^{eL}



CryptoMiniSAT5 [3], which is known to be a strong system for solving cryptographic benchmarks. This system solves 41 instances, while two top `expSAT` solvers on this benchmark MplLRB^{eL} and MplCBT^{eLV}, solve 46 and 44 instances, respectively. Figure 5.9 shows the number of solved instances as a function of time for these three solvers, where both of these `expSAT` extensions scales slightly better than CryptoMiniSAT5.

5.6 Related Work

Randomized exploration in SAT is used in local search methods such as GSAT [71] and WalkSAT [70]. The *Satz* algorithm [40] heuristically selects a variable x, then performs two separate unit propagations with x and $(\neg x)$ respectively, in order to evaluate the potential of x. Modern CDCL SAT solvers include exploration components. For example, in MiniSAT, a small percentage of variables is selected randomly [23]. CaDiCaL, a more advanced CDCL SAT solver, triggers random-walk based local search in regular intervals to determine better polarity values for the variables [12].

In contrast to the DPLL and CDCL SAT frameworks, which employ depth-first

[3]Source: https://www.msoos.org/cryptominisat5/

search, in [58] the authors propose a SAT solver UCTSAT which employs MCTS with UCT (Upper Confidence bounds applied to Trees). Given a SAT instance, UCTSAT repeatedly invokes UCT search at the root and incrementally builds a SAT search tree based on the value estimates of the search states. The value of a state is estimates come from the outcomes of random samples of states that were visited in previous iterations.

Liang et al. [41] propose a look-ahead based branching heuristic that greedily maximizes the GLR score. Compared to this work we perform nondeterministic exploration of the search space with a small subset of unassigned variables per random walk, and prioritize variables that generate high-quality conflicts. Since decision time is disregarded in their work, there is no basis to compare the experimental results.

In [9], the authors propose a hybrid SAT solver SATHYS, which combines a CDCL SAT solver with a local search SAT solver. The search alternates between these two solvers, and they exchange information. The local search solver helps CDCL by identifying the most promising literal assignment to branch on, while CDCL guides the local search out of local minima.

The Conflict History-based Branching (CHB) [42] and Learning Rate Based (LRB) [43] heuristics model variable selection as a Multi-Armed Bandit (MAB) problem, which is solved using the Exponential Recency Weighted Average (ERWA) algorithm. Both of these heuristics compute rewards from the conflict history of unassigned variables, in order to rank them.

In [5], the authors study the search process of Glucose by computing four measures over a fixed set of benchmark instances to derive insights into the search process of Glucose. One of these measures is *decisions per conflict*, which approximates the average decision distance between two conflicts in a given run of a solver with a given SAT instance. In expSAT, we utilize this notion to define substantial CD phases.

Cai 2021 et al. [16] present a hybrid SAT solving algorithm that alternates between local search and CDCL search. During CDCL search, this algorithm attempts to identify a promising branch, which may contain a satisfying assignment.

Once such a promising branch is identified, a local search episode is triggered. This episode attempts to find a total assignment by extending the current partial assignment, where it repeatedly flips the remaining unassigned variables to satisfy the currently unsatisfied clauses. If the problem is not solved before reaching a predefined number of flips, it returns back to CDCL search. Before starting the CDCL search again, it updates the VSIDS and LRB scores of those variables, flipping of which created conflicts during the last local search episode. Similar to their work, we perform local search during in an exploration episode and update the variable selection heuristics. However, contrary to the work of [16], which performs local search to find satisfiable assignments, in `expSAT`, amid a CD phase, we perform local search during an exploration episode to identify conflict friendly variables.

5.7 Conclusions

In Chapter 4, we provided empirical insights into the conflict generation pattern in CDCL SAT solving. Our case studies with two leading conflict driven CDCL branching heuristics `VSIDS` and `LRB` provide empirical evidence that conflicts are generated in bursts, often followed by long conflict depression phases. The latter occur due to weaknesses of the underlying branching heuristics.

This observation led us to develop an exploration-guided CDCL SAT solver framework called `expSAT`, which performs random exploration amid substantial conflict depression phases. Under the `expSAT` framework, we developed `expVSIDS` and `expLRB`, which combine the exploration scores from random walks with `VSIDS` and `LRB` scores.

Our extensive empirical evaluation of the `expSAT` approach with five state-of-the-art CDCL SAT solver results in 11 `expSAT` extensions with `expVSIDS` and `expLRB`. We show strong performance gains with most of our extensions for instances from SAT Competition 2017 and 2018, and especially for 52 hard SATCoin benchmarks. An analysis of the experimental data shows that `expSAT` solvers increase GLR, and reduce average LBD and average CD phase lengths in most cases. This explains the gains with the `expSAT` approach. Our in-depth discussion on various aspects of CD phases and `expSAT` reveal interesting insights for CDCL SAT solving.

Chapter 6

Conflict Generating Decisions in CDCL

In [41], the authors have shown empirically that the most efficient CDCL decision heuristics, such as VSIDS, LRB and CHB, produce about 0.5 conflicts per decision, on average. Each decision step in CDCL can generate 0, 1 or more than one conflict.

Conflicts play a crucial role in CDCL search. A better understanding of conflict generating decisions is a step towards a better understanding of CDCL and may open up new directions to improve CDCL search. Motivated by this, in this Chapter, we study conflict producing decisions in CDCL.

We categorize each conflict-producing decision as a *single conflict* (`sc`) or a *multi-conflicts* (`mc`) decision, depending on whether it produces one, or more than one conflict. We label the resulting learned clauses `sc` and `mc` clauses accordingly.

The contributions of this Chapter are:

1. We compare `sc` and `mc` decisions in terms of the average quality of the learned clauses. The average LBD score is significantly lower (of better quality) for `sc` than for `mc` clauses.

2. We analyze the distribution of conflicts in `mc` decisions. Although a `mc` decision can produce a large number of consecutive conflicts, `mc` decisions with a low number of consecutive conflicts are more frequent.

3. We contribute an analysis that shows how consecutive clauses learned by a `mc` decision are connected to each other.

4. We introduce the measure of `ConflictProximity` to study the relation between conflicts in a given conflict sequence. The proximity between a set of conflicts is defined in terms of literal blocks that are shared between the reason clauses of these conflicts. We show that the conflicts which are discovered during the same `mc` decision are closer by this measure than conflicts discovered by consecutive `sc` decisions.

5. We develop the CDCL decision strategy *Common Reason Variable Reduction* (`CRVR`), which reduces the priority of some variables that appear in `mc` clauses. Our empirical evaluation of `CRVR` on benchmarks from the main-track of SAT Competition-2020 (**TS3**) shows performance gains for satisfiable instances in several leading solvers.

6.1 Notation

We denote a CDCL solver ψ running a given SAT instance F by ψ_F. Assume that this run makes d decisions and generates c conflicts and learns a set of learned clause $\mathcal{L}$.

Single Conflict (`sc`) and Multi-conflicts (`mc`) Decisions

A `sc` decision generates exactly one conflict and learns a `sc` clause, while a `mc` decision generates more than one conflict and accordingly learns multiple `mc` clauses. Let ψ_F take s `sc` decisions and m `mc` decisions, learning the set of clauses $\mathcal{L}_s$ and $\mathcal{L}_m$, respectively. Then $d = m + s$ and $|\mathcal{L}| = |\mathcal{L}_s \cup \mathcal{L}_m|$.

Burst of `mc` Decisions

We define the `burst` of a `mc` decision as the number of conflicts (i.e., learned clauses) generated within that `mc` decision. Let b_i is the burst of i^{th} `mc` decision. We have $|\mathcal{L}_m| = \sum_{i=1}^{m} b_i$.

For ψ_F, we define

- `avgBurst`, the average burst over *m* `mc` decisions as $\frac{|L_m|}{m}$

- maxBurst, the maximum burst among the bursts of mc decisions as
$$\mathbf{max}_{i=1\ldots m}\ b_i$$
- We define a mapping $\texttt{count}_b : \mathbb{Z} \longmapsto \mathbb{Z}$ which takes a burst $b \geq 2$ as input and outputs the number of mc decisions with burst b.

Learned Clause Quality Over sc and mc Decisions

Let $lbd(L)$ be the LBD score of a learned clause L. For ψ_F, we define

- the average LBD score aLBD over $\mathcal{L}$ learned clauses as $\frac{\sum_{L\in\mathcal{L}} lbd(L)}{|\mathcal{L}|}$.
- the average LBD score $\texttt{aLBD}_{\texttt{sc}}$ over $\mathcal{L}_s$ learned clauses as $\frac{\sum_{L_s\in\mathcal{L}_s} lbd(L_s)}{|\mathcal{L}_s|}$.
- the average LBD score $\texttt{aLBD}_{\texttt{mc}}$ over $\mathcal{L}_m$ learned clauses as $\frac{\sum_{L_m\in\mathcal{L}_m} lbd(L_m)}{|\mathcal{L}_m|}$.

For an mc decision $\mathcal{M}$, we denote the minimum LBD score among its learned clauses by $\texttt{min\_LBD}_{\mathcal{M}}$. For ψ_F, $\texttt{avg\_min\_LBD}_{\texttt{mc}}$ is the average minimum LBD over m mc decisions.

6.2 An Empirical Analysis of sc and mc Decisions

In this section, we present our empirical study of conflict-generating decisions in CDCL search. We use MplDL as CDCL solver and investigate its sc and mc decisions in the test set **TS3**.

Table 6.1: Conflict Generating Decisions. Columns A to G shows average measures over the number of instances shown in the Count column.

Type	**Count**	**Conflict Frequency**		**Clause Quality**			**mc Bursts**	
		A: PDSC	B: PDMC	C: $\texttt{aLBD}_{\texttt{sc}}$	D: $\texttt{aLBD}_{\texttt{mc}}$	E: $\texttt{avg\_min\_LBD}_{\texttt{mc}}$	F: avgBurst	G: maxBurst
SAT	106	6%	**10%**	22.32	32.30	**18.90**	2.69	33.76
UNSAT	110	7%	**12%**	236.26	389.68	**80.80**	2.70	52.37
UNSOLVED	184	9%	**16%**	**72.14**	144.75	73.38	2.60	29.70
Combined	400	8%	**13%**	80.68	104.07	**60.86**	2.65	94.51

6.2.1 Distributions of sc and mc decisions

We denote *Percentage of Decisions with Single Conflict* and *Percentage of Decisions with Multiple Conflicts* as PDSC and PDMC, respectively. Columns A and B in Table 6.1 show the average PDSC and PDMC values for the test instances from **TS3**, under SAT, UNSAT and UNSOLVED. Overall, 8% of all decisions are `sc` and 13% are `mc` (see the bottom row). On average, about 21% (8%+13%) of the decisions are conflict producing, almost two thirds of all conflict producing decisions are `mc`.

Since `mc` decisions produce 2.65 (Column F) conflicts on average, this generates almost 1 conflict per 2 decisions, which is reflected in the average GLR value of 0.49 for these instances.

6.2.2 Learned Clause Quality in sc and mc Decisions

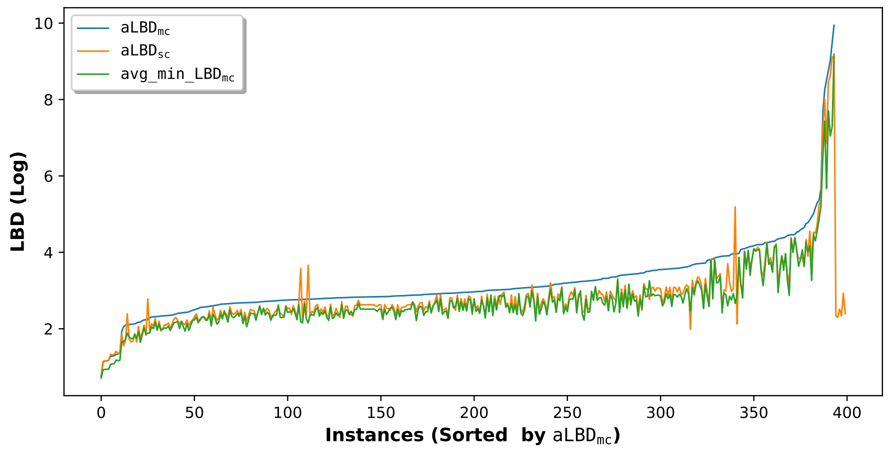


Figure 6.1: Clause Quality in Conflict Generating Decisions; LBD scores are shown in Log (natural) scale

Columns C and D in Table 6.1 compare LBD scores of `sc` and `mc` decisions. On average, `sc` decisions generate higher quality learned clauses (with lower LBD scores). However, Column E shows that in most cases, the average minimum LBD score over the clauses in a single `mc` decision is lower than for `sc`. The exception is the UNSOLVED category. Fig. 6.1 shows per-instance details of these three measures in a log scale. In almost all instances, LBD scores for `mc` (blue) are higher than for `sc` (orange), and minimum `mc` LBD (green) is lowest.

To summarize, on average `mc` decisions are conflict-inefficient compared to `sc` decisions. However, on average the best quality learned clause from a `mc` decision has better quality than the quality of a `sc` clause.

6.2.3 Bursts of mc Decisions

Column F in Table 6.1 shows the average value of `avgBurst` for the test set **TS3**. On average, the `burst` of `mc` decisions are quite small, about 2.65. However, as shown in column G, the average value of `maxBurst` is very high. The left plot in Fig. 6.2 compares these values for each test instance in log scale. In almost all cases `maxBurst` (orange) is much larger than the average (blue). This indicates that while large bursts of `mc` decisions occur, they are rare, as indicated by the average of 2.65. In the following section, we analyze the distribution of `mc` bursts in details.

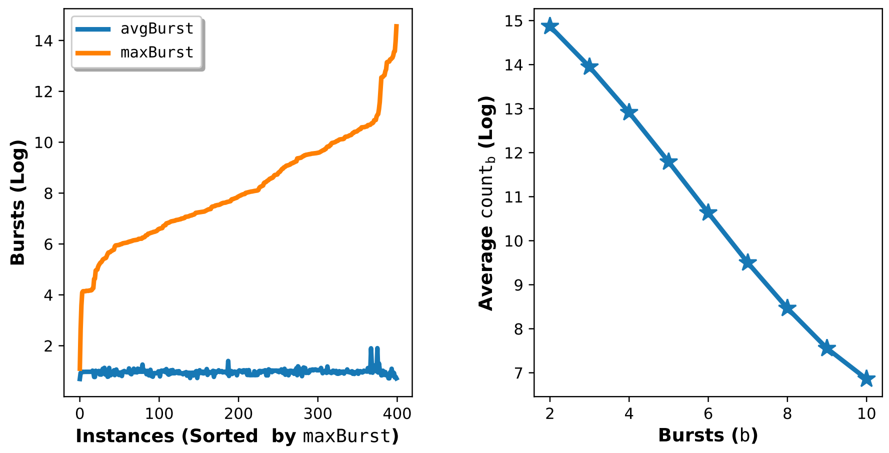


Figure 6.2: Analysis of Bursts of `mc` decisions; Bursts are shown Log (natural) scale.

Distribution of mc Decisions by Burst Size

Column G of Table 6.1 illustrates that `maxBurst` can be very large. To simplify our quantitative analysis we focus on counting `mc` decisions with bursts up to 10. The plot on the right of Fig. 6.2 shows the average (over the 400 instances in our test set) count (in Log scale) of the number of bursts of a given size b, with $2 \leq b \leq 10$. The frequency of bursts decreases exponentially with their size.

6.3 Clause Learning in `mc` Decisions

In this section, we establish a structural property of the learned clauses in `mc` decisions.

Formalization of `mc` Decisions

Let v be the decision variable for the `mc` decision $\mathcal{M}$ with `burst` $x \geq 2$. At the time when the search reaches the first conflict C_1 in $\mathcal{M}$, let P_0 be the set of literal assignments that followed the assignment of v. With $1 \leq i \leq x$, let C_i be the i^{th} conflicting clause, from which the clause $L_i = R_i \vee \{\neg f_i\}$ is learned. Here, R_i is the reason clause and f_i is the fUIP literal for this i^{th} conflict. After learning L_i, and after backtracking, $\neg f_i$ is the only unassigned literal in L_i, and it is immediately unit-propagated from L_i. Let P_i be the propagation block that contains literal assignments starting from the assignment of $\neg f_i$ until the search reaches the conflicting clause C_{i+1}. Let $\mathcal{L} = (L_1, \ldots, L_x)$ be the ordered sequence of x learned clauses in $\mathcal{M}$.

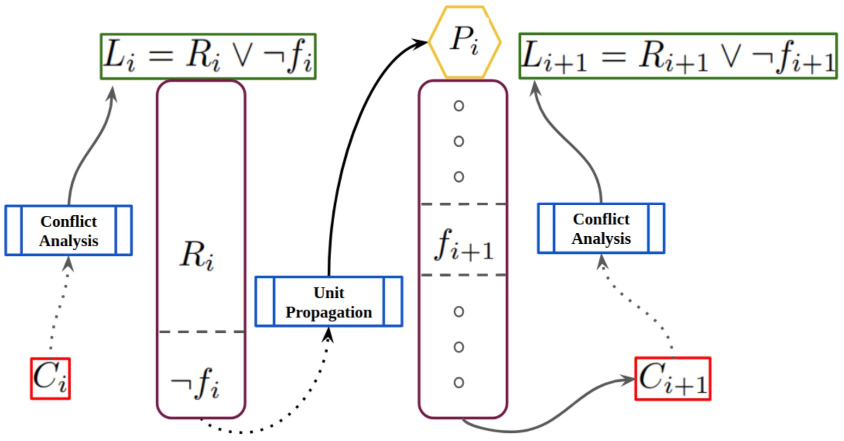


Figure 6.3: Connection between learned clauses in $\mathcal{M}$. After backtracking, L_i forces $\neg f_i$, the negated literal of the fUIP for the conflict at C_i. This forced assignment creates a block of assignments P_i, until the search reaches a conflict at C_{i+1}. P_i contains f_{i+1}, which is the fUIP of the conflict at C_{i+1}. From C_{i+1}, Conflict Analysis learns L_{i+1}, which contains $\neg f_{i+1}$.

Claim 1: *$\mathcal{M}$ learns a sequence of clauses $\mathcal{L} = (L_1, \ldots, L_x)$, where a clause L_i $(1 \leq i < x)$ implicitly constructs L_{i+1}, by implying f_{i+1}, the fUIP literal for the $(i+1)^{th}$ conflict, from which L_{i+1} is learned.*

We justify Claim 1 as follows:

With $1 \leq i < x$, after learning the i^{th} clause $L_i = R_i \vee \neg f_i$ and backtracking to a previous level, the literal $\neg f_i$ (the negated literal of the fUIP of i^{th} conflict) is forced in L_i. This forced assignment creates a propagation block P_i and reaches the conflicting clause C_{i+1}. From C_{i+1} the search learns the next clause $L_{i+1} = R_{i+1} \vee \neg f_{i+1}$ within the current mc decision. Clearly, the fUIP of $i + 1^{th}$ conflict, $f_{i+1} \in P_i$, as $\neg f_{i+1} \in L_{i+1}$ is the only literal assigned in the current decision level. Fig. 6.3 shows the connection between L_i and L_{i+1}.

We have $(a)(L_i = R_i \vee \neg f_i) \rightarrow f_{i+1}, \quad (b) f_{i+1} \in P_i, \quad$ and $\; (c) \neg f_{i+1} \in L_{i+1}$

Hence, under the current partial assignment, the learning of L_i is a *sufficient condition* for the learning of L_{i+1}. Any pair of consecutive clauses (L_i, L_{i+1}) are connected via the pair of assignments $(\neg f_i, f_{i+1})$, where the first assignment in this pair is the negated literal of the fUIP literal for the i^{th} conflict and the second assignment is the fUIP literal for the $(i+1)^{th}$ conflict.

Since the argument applies to all $1 \leq i < x$, we have the desired result.

6.4 Proximity between Conflicts Sequences in CDCL

By Claim 1, we see that learned clauses in a mc decision are connected in a specific way. This indicates that conflicts in a mc decision are also related, as clauses are learned from conflicts. Here, we first introduce the measure of ConflictProximity to study proximity between conflict sequences and then present an empirical study to reveal insights on proximity between conflicts sequences in CDCL.

6.4.1 Conflict Proximity

The notion of conflict proximity uses a novel measure called *Literal Block Proximity*, which measures the commonality of literal blocks between a sequence of *reason clauses* over a sequence of conflicts.

Literal Block Proximity

Assume that from a conflicting clause C, $L = R \vee \neg f$ is learned, where R is the reason clause for the conflict at C and f is the fUIP literal of the current conflict. We define a mapping

$$\mathcal{D} : \texttt{Clause} \longmapsto \{\texttt{dl}_1 \ldots \texttt{dl}_\texttt{n}\}$$

which maps a given reason clause R to the set of distinct decision levels in R. Each $\texttt{dl} \in \mathcal{D}(R)$ corresponds to the block of literals $\texttt{block}_\texttt{dl}$ in R which were assigned in $\texttt{dl}$.

Let $\mathcal{R}_\mathcal{C} = (R_1, \ldots, R_m)$ be the sequence of reason clauses for the conflicting clauses in $\mathcal{C} = (C_1, \ldots C_m)$, where $R_i \in \mathcal{R}$ is the reason clause for the conflict at $C_i \in \mathcal{C}$. We define the set *Literal Block Proximity* (LBP) for $\mathcal{R}_\mathcal{C}$, $\mathbf{LBP}_{\mathcal{R}_\mathcal{C}}$ by

$$\mathbf{LBP}_{\mathcal{R}_\mathcal{C}} = \mathcal{D}(R_1) \cap \cdots \cap \mathcal{D}(R_m)$$

That is, $\mathbf{LBP}_{\mathcal{R}_\mathcal{C}}$ is the set of decision levels that are common in all clauses in $\mathcal{R}$. Therefore, the assignments in $\texttt{block}_\texttt{dl}$ with $\texttt{dl} \in \mathbf{LBP}_{\mathcal{R}_\mathcal{C}}$ contribute to the discovery of every conflicting clause in $\mathcal{C}$.

Example 1: *Let $\mathcal{R}_\mathcal{C} = (R_a, R_b)$ be a set of reason clauses for the conflicts at clauses in $\mathcal{C} = (C_a, C_b)$. Let $\mathcal{D}(R_a)$ = {2, 9, 14, 35, 110} and $\mathcal{D}(R_b)$ = {9, 10, 11, 35, 98, 110} be the sets of decision levels in R_a and R_b, respectively. Then $\mathbf{LBP}_{\mathcal{R}_\mathcal{C}} = \mathcal{D}(R_a) \cap \mathcal{D}(R_b)$ = {9, 35, 110}. The assignments in $\texttt{block}_9$, $\texttt{block}_{35}$, and $\texttt{block}_{110}$ contribute to the generation of conflicts in both C_a and C_b.*

Conflict Proximity in a Conflict Sequence

For a reason clause sequence $\mathcal{R}_\mathcal{C} = (R_1, \ldots, R_m)$, we define the `ConflictProximity` $\texttt{cp}_{\mathcal{R}_\mathcal{C}}$, with $0 \leq \texttt{cp}_{\mathcal{R}_\mathcal{C}} \leq 1$ as

$$\texttt{cp}_{\mathcal{R}_\mathcal{C}} = \frac{|\mathbf{LBP}_{\mathcal{R}_\mathcal{C}}|}{|U_{\mathcal{R}_\mathcal{C}}|}$$

where $U_{\mathcal{R}_\mathcal{C}} = \mathcal{D}(R_1) \cup \cdots \cup \mathcal{D}(R_m)$ is the set of all literal blocks in $\mathcal{R}_\mathcal{C}$. In Example 1, $U_{\mathcal{R}_\mathcal{C}}$ = {2, 9, 10, 11, 14, 35, 98, 110} and $\texttt{cp}_{\mathcal{R}_\mathcal{C}} = \frac{|\mathbf{LBP}_{\mathcal{R}_\mathcal{C}}|}{|U_{\mathcal{R}_\mathcal{C}}|} = 3/8$.

For any two given reason clause sequences $\mathcal{R}_{\mathcal{C}p}$ and $\mathcal{R}_{\mathcal{C}q}$, with $|\mathcal{R}_{\mathcal{C}p}| = |\mathcal{R}_{\mathcal{C}q}|$, if $\texttt{cp}_{\mathcal{R}_{\mathcal{C}p}} > \texttt{cp}_{\mathcal{R}_{\mathcal{C}q}}$, then we call conflicts associated with the reason clauses in $\mathcal{R}_{\mathcal{C}p}$ are more closely related to each other than conflicts associated with the reason clauses in $\mathcal{R}_{\mathcal{C}q}$.

We now study proximity of conflicts in CDCL under `ConflictProximity`.

6.4.2 Proximity of Conflicts over **sc** and **mc** Decisions

While the learned clauses in a `mc` decision are connected, each learned clause in a `sc` decision is learned in isolation. We propose the following hypothesis:

Hypothesis 1: *On average, conflicts in a* `mc` *decision with burst x are more closely related than conflicts which are generated in the last x* `sc` *decisions.*

We support this hypothesis by comparing the `ConflictProximity` of reason clauses over `mc` and `sc` decisions.

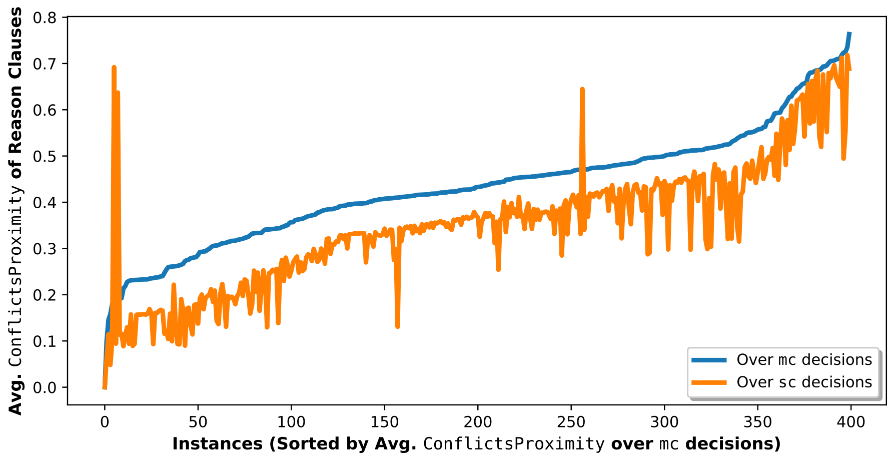


Figure 6.4: Comparison of average `ConflictProximity` of reason clauses over `mc` and `sc` Decisions

Experiment

We have performed an experiment with 400 instances from **TS3** with `MplDL` with a time limit of 5,000 seconds. For each run of an instance, whenever the search finds a `mc` decision with burst x, we compute the LBP and `ConflictProximity` for the reason clauses (i) for these x conflicts and (ii) for the last x conflicts in `sc` decisions. For this experiment, we collect data for bursts $x \leq 10$. For a run with an instance, we compute the average of `ConflictProximity` of reason clauses separately over `mc` and `sc` decisions.

Fig. 6.4 shows that average `ConflictProximity` for the reason clauses over `mc` decisions (blue lines, average is 0.43) are higher than `ConflictProximity` of reason clauses over `sc` decisions (orange line, average is 0.34) for almost all instances. This validates Hypothesis 1.

6.5 The Common Reason Variable Reduction Strategy

6.5.1 Common Reason Decision Variables

Assume that a `mc` decision $\mathcal{M}$ finds $x \geq 2$ consecutive conflicts within its decision. Let $\mathcal{R} = (R_1, \ldots, R_x)$ be the sequence of reason clauses for these x conflicts. $\mathbf{LBP}_{\mathcal{R}}$ is the set of common decision levels over $\mathcal{R}$. For each decision level `dl` $\in \mathbf{LBP}_{\mathcal{R}}$, we call `dl`, a *common reason decision level* and the decision variable v at `dl`, a *common reason decision variable (CRV)* for $\mathcal{M}$. If $|\mathbf{LBP}_{\mathcal{R}}| = z$, then there are z CRVs in $\mathcal{M}$. The CRVs in $\mathcal{M}$ are the decision variables from previous decision levels, which contributed to the generation of all the conflicts in $\mathcal{M}$.

6.5.2 Poor `mc` Decisions

Recall that in Section 6.2 (Fig. 6.1), we observed that on average, `mc` decisions (blue line) produce lower quality clauses than `sc` decisions (orange line). However, the best quality clause (green line) in a `mc` decision has better average quality than other learned clauses. Nevertheless, in a `poor mc` decision its best quality learned clause is worse than the average quality.

Definition 13: *(Poor `mc` Decision) A `mc` decision $\mathcal{M}$ is `poor` if the quality of the best learned clause in $\mathcal{M}$ is lower than a dynamically computed threshold θ, the average quality of the last k learned clauses.*

Table 6.2: The CRVR Decision Strategy

```
Procedure DetectPoorCRV
1. $\theta \leftarrow aLBD(k)$
2. if (min_LBD$_{\mathcal{M}}$ $> \theta$)
   a. $\mathcal{R} \leftarrow reason\_clauses\_in\_\mathcal{M}()$
   b. for each dl $\in \mathbf{LBP}_{\mathcal{R}}$
      i. v $\leftarrow dvar($dl$)$
      ii. poor_crv[v] $\leftarrow$ true
```

```
Procedure CRVRBranching
1. selected $\leftarrow$ false
2. while (not selected)
   a. y $\leftarrow$ select_next_free_var()
   b. if (poor_crv[y])
      i. activity[y] $\leftarrow$ activity[y] * (1-Q)
      ii. poor_crv[y] $\leftarrow$ false
      iii. reorder()
   c. else
      i. selected $\leftarrow$ true
```

6.5.3 The CRVR Decision Strategy

We summarize the previous two subsections as follows:

- Conflicts in a poor mc decision are not likely to be helpful, as the quality of its best learned clause is lower than the recent search average.

- The CRVs in a poor mc decision combinedly contribute to the generation of these conflicts.

Does suppression of such CRVs for future decisions help the search? We design a decision strategy named *common reason variable score reduction* (CRVR), which can be integrated with any activity based variable selection decision heuristics such as VSIDS and LRB. The high-level idea of CRVR is as follows: Once a poor mc decision is detected, CRVR (i) finds the CRVs for that poor mc and (ii) marks those CRVs as poor CRVs, (iii) then reduces the activity scores of those poor CRVs for future decisions. CRVR consists of two procedures, DetectPoorCRV and CRVRBranching. Pseudo-codes of these two procedures are shown in Table 6.2.

DetectPoorCRV

This procedure is invoked at the end of an mc decision $\mathcal{M}$. It computes a dynamic conflict quality threshold θ, the average LBD score of the last k learned clauses. Then it determines if $\mathcal{M}$ is poor by comparing min_LBD$_{\mathcal{M}}$with θ. In this case, DetectPoorCRV obtains the sequence of reason clauses $\mathcal{R}$ in $\mathcal{M}$ and computes

$\mathbf{LBP}_{\mathcal{R}}$. For each decision level dl $\in \mathbf{LBP}_{\mathcal{R}}$, any decision variable v at dl is marked as poor.

CRVRBranching

Pseudocode for CRVRBranching is shown in the right side of Table 6.2. This procedure modifies a CDCL decision routine to lazily reduce the activity score of poor CRVs. It employs a while loop until a variable is selected, where in each iteration of the loop, it performs the following operations: (i) obtain a free variable y, where y is the free variable with largest activity score. (ii) check if y is marked as poor. If y is poor, then it reduces activity[y], by a factor of Q, a user defined parameter with $0 < \mathtt{Q} < 1$. (iii) unmark y to no longer to be poor and reorders the variables by their activity scores. The reduction of the activity score of y, followed by reordering, decreases the selection priority of y.

6.6 Experimental Evaluation

6.6.1 Implementation

We implemented CRVR in three leading baseline solvers MplDL, Kissat-sat and Kissat-default. We call the extended solvers MplDL$^{\mathbf{crvr}}$, Kissat-sat$^{\mathbf{crvr}}$, and Kissat-default$^{\mathbf{crvr}}$, respectively. The solver MplDL employs a combination of the decision heuristics DIST [80], VSIDS [49] and LRB [43], which are activated at different phases of the search, whereas Kissat-sat and Kissat-default use VSIDS and Variable Move to Front (VMTF) [62] alternately during the search.

The heuristics DIST, VSIDS and LRB share similar data structures: all maintain an *activity score* for each variable. Whenever a variable is involved in a conflict, its activity score is increased. In contrast, VMTF maintains a queue of variables, where a subset of variables appearing in a learned clause are moved to the front of that queue in an arbitrary order.

CRVR works on top of activity-based decision heuristics. Hence in Kissat-sat$^{\mathbf{crvr}}$ and Kissat-default$^{\mathbf{crvr}}$, we employ CRVR only in phases when VSIDS is active.

In all of our extended solvers, we use the following parameter values: a length of window of recent conflicts $k = 50$ and an activity score reduction factor $\texttt{Q} = 0.1$.

Table 6.3: Comparison between 3 baselines and their `CRVR` extensions with $\mathbf{TS3}$ instances.

Systems	**SAT**	**UNSAT**	**Combinded**	**PAR-2**
`MplDL`	106	**110**	216	2065
`MplDL`$^{\texttt{crvr}}$	**116 (+10)**	107 (-3)	**223 (+7)**	**2001**
`Kissat-sat`	148	**118**	**266**	**1552**
`Kissat-sat`$^{\texttt{crvr}}$	**150 (+2)**	114 (-4)	264 (-2)	1565
`Kissat-default`	134	**126**	260	1624
`Kissat-default`$^{\texttt{crvr}}$	**139 (+5)**	125 (-1)	**264 (+4)**	**1588**

6.6.2 Experiments and Results

We conduct experiments with the same set of 400 instances from $\mathbf{TS3}$ with a 5,000 seconds timeout per instance. We compare the `CRVR` extensions and their counterpart baselines in terms of number of solved instances, solving time and PAR-2 score.

Table 6.3 compares `MplDL`$^{\texttt{crvr}}$, `Kissat-sat`$^{\texttt{crvr}}$, and `Kissat-default`$^{\texttt{crvr}}$ with their baselines. All extensions show performance gains on SAT instances, but lose on UNSAT instances. The strongest gain is for `MplDL`$^{\texttt{crvr}}$, which solves 10 additional SAT instances, but 3 less UNSAT instances, for an overall gain of 7 instances. `Kissat-sat`$^{\texttt{crvr}}$ solves 2 more SAT instances, but loses 4 UNSAT instances, an overall loss of 2. `Kissat-default` solves 5 more SAT instances, and 1 fewer UNSAT instance, +4 overall.

The PAR-2 results are consistent with the solution counts. While `Kissat-sat`$^{\texttt{crvr}}$ has a slightly worse 0.8%) PAR-2 score compared to `Kissat-sat`, both `MplDL`$^{\texttt{crvr}}$ and `Kissat-default`$^{\texttt{crvr}}$ have significantly lower PAR-2 scores (by 3.19% and 2.28%, respectively), which reflects overall better performance of these two systems.

Fig. 6.5 compares the relative solving speed of `MplDL`$^{\texttt{crvr}}$ (blue), `Kissat-sat`$^{\texttt{crvr}}$ (orange), and `Kissat-default`$^{\texttt{crvr}}$ (green) against their baselines by plotting the difference in the number of instances solved as a function of time. `MplDL`$^{\texttt{crvr}}$

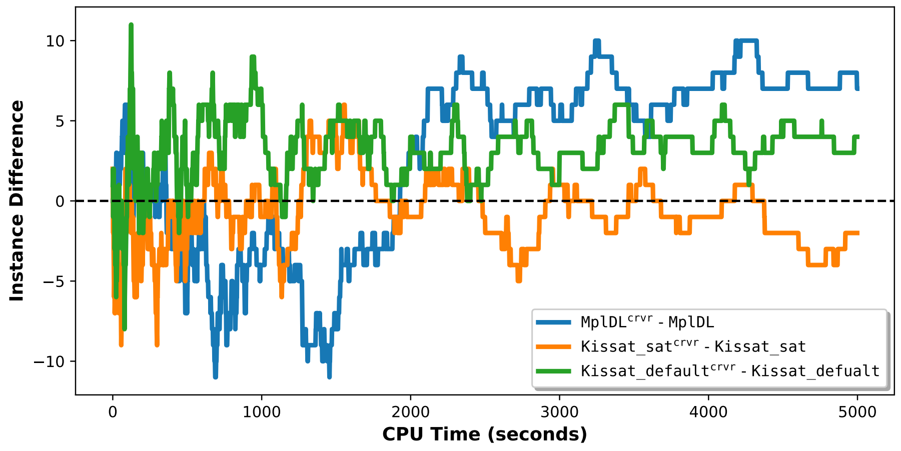


Figure 6.5: Solve time comparisons. For any point above 0 in the vertical axis, our extensions solve more instances than their baselines at the time point in the horizontal axis.

(blue) performs slightly worse than `MplDL` early on, but beats the baseline consistently after 1,900 seconds. `Kissat-default`$^{\mathtt{crvr}}$ (green) is ahead of its corresponding baseline `Kissat-default` at most time-points. Compared to `Kissat-sat`, `Kissat-sat`$^{\mathtt{crvr}}$ (orange) is behind at most time points.

Overall, compared to their baselines, our extensions perform better on SAT instances, but lose a small number of UNSAT instances. We discuss this behavior in the next section.

6.7 Discussion of the Experimental Results

6.7.1 Detailed Performance Analysis of CRVR

Recall that for a run of a given solver, the metric GLR measures the overall conflict generation rate of the search, average LBD (aLBD) measures the average quality of the learned clauses and G2L measures the fraction of learned clauses which are glue. We relate the performance of `CRVR` with these three metrics. We consider two subsets of instances from **TS3**, where `MplDL` and `MplDL`$^{\mathtt{crvr}}$ show opposite strengths:

- `CRVR–bad`: 12 instances solved by `MplDL`, but not by `MplDL`$^{\mathtt{crvr}}$.

- `CRVR–good`: 19 instances solved by `MplDL`$^{\text{crvr}}$, but not by `MplDL`.

Table 6.4: Relating solving efficiency with the three metrics with the average GLR, average LBD and average G2L. More efficient branching heuristics tend to achieve higher average GLR [41], higher average G2L [18] and lower average aLBD [41] than less efficient ones.

Instance Sets	**Count (SAT+UNSAT)**	**average GLR**		**average aLBD**		**average G2L**	
		MplDL	**MplDL**$^{\text{crvr}}$	**MplDL**	**MplDL**$^{\text{crvr}}$	**MplDL**	**MplDL**$^{\text{crvr}}$
`CRVR–good`	19 (18+1)	**0.58**	53	4399.91	**147.69**	0.003	**0.018**
`CRVR–bad`	12 (8+4)	0.56	0.56	18.83	**18.53**	**0.024**	0.023

The 19 instances in `CRVR–good` (first row of Table 6.4) are solved exclusively by `MplDL`$^{\text{crvr}}$. For this subset, `MplDL`$^{\text{crvr}}$ learns clauses at a slightly lower rate. However, for `CRVR–good`, the average aLBD (resp. average G2L) is significantly lower (resp. higher) with `MplDL`$^{\text{crvr}}$. `CRVR` helps to (i) learn higher quality clauses, and (ii) learn more glue clauses relative to the number of clauses for the subset of instances in `CRVR–good`, for which `MplDL`$^{\text{crvr}}$ is efficient.

18 of the 19 instances in `CRVR–good` are SAT. The learning of significantly better quality of clauses with `MplDL`$^{\text{crvr}}$ for these SAT instances may just explain the good performance of `CRVR` on SAT instances.

For the 12 instances in `CRVR–bad` (second row of Table 6.4), `MplDL` learns clauses at the same rate, but learns clauses which are of slightly lower quality than the clauses learned by `MplDL`$^{\text{crvr}}$. However, for this set, the average G2L value is slightly higher with `MplDL` than `MplDL`$^{\text{crvr}}$. This could explain the better performance of `MplDL` for this subset.

6.7.2 Performance behavior of CRVR on SAT instances

The author of [54] showed that Glucose with LBD core cut 2 (clauses with LBD score upto 2 never get deleted) is as powerful as original Glucose (which retains some learned clauses with LBD score > 2) for SAT instances, while performance of the modified Glucose on UNSAT instances is worse (Table 2.1 in [54], page 51). This indicates that retaining glue-clauses has more positive impact on SAT instances than UNSAT instances.

We have analyzed data for `Mp1DL` and `Mp1DL`$^{\texttt{crvr}}$ for the experiments in Table 6.3. The G2L value is slightly higher for `Mp1DL`$^{\texttt{crvr}}$ than `Mp1DL`. Further, as shown in the previous section, the gap in G2L value between these two systems is highly pronounced for the 19 instances, 18 of which are SAT, which are *exclusively* solved by `Mp1DL`$^{\texttt{crvr}}$, but not by `Mp1DL`. For these 19 instances, the average G2L value for `Mp1DL`$^{\texttt{crvr}}$ (0.018) is 3 times higher for than `Mp1DL` (0.006). Hence the propensity of `Mp1DL`$^{\texttt{crvr}}$ to learn more glue clauses relative to the total learned clauses may help the solver in solving more SAT instances.

6.8 Related Work

Audemard and Simon [5] briefly studied *decisions with successive conflicts*, which we refer to as `mc` decisions in this paper. They studied the number of successive conflicts in the CDCL solver Glucose on a fixed set of instances. Here, we present a more formal and in-depth study of `mc` decisions. The authors of [41] relate conflict generation propensity and learned clause quality with the efficiency of several decision heuristics. In contrast, we study and compare the conflict quality of two types of conflict producing decisions for CDCL. The conflict generation pattern in CDCL is studied in [17], showing that CDCL typically alternates between bursts and depression phases of conflict generation. While that work presented an in-depth study of the conflict depression phases in CDCL, here we study the conflict bursts phases in detail. The authors of [18] studied the conflict efficiency of decisions with two types of variables: those that appear in glue clauses and those that do not. In this Chapter, we compare the conflict efficiency of conflict producing decisions.

6.9 Conclusions

We presented a characterization of `sc` and `mc` decisions in terms of average learned clause quality that each type produces. Then we analyze how `mc` decisions with different bursts are distributed in CDCL search. Our theoretical analysis shows that learned clauses in a `mc` are connected, indicating that conflicts that occur in a `mc` decision are related to each other. We introduced a measure `ConflictProximity` that enables the study of proximity of conflicts in a given sequence of conflicts. Our empirical analysis shows that conflicts are more *closely related* in `mc` decisions than in consecutive `sc` decisions. Finally, we formulated a novel CDCL strategy `CRVR` that reduces the activity score of some variables that appear in the clauses learned over `mc` decisions. Our empirical evaluation with three modern CDCL SAT solvers shows the effectiveness of `CRVR` for the SAT instances from **TS3**.

Chapter 7

Conclusion

In this thesis, we have studied CDCL SAT solving algorithms empirically. Particularly, we focused on heuristic guided variable selection steps in CDCL to discover their impact on conflict generation and clause learning. Our studies have unveiled a series of important insights, some of which lead to the developments of novel techniques that extend the standard CDCL SAT solving algorithms. Empirical evaluations of these new techniques reveal their effectiveness on state-of-the-art CDCL solvers over a diverse set of benchmarks. Our analysis of the experimental results from these evaluations reveals another set of interesting insights, some of which are novel for CDCL SAT search.

In this chapter, first, we present an overview of the main results of this thesis work. Next, we present the potential impact of this work. We conclude this chapter by laying out some future work of this thesis.

7.1 Overview of this Thesis

Here we present a brief overview of this thesis work.

- **Chapter 3**: We relate variable selection step in CDCL with glue clause, a special type of learned clause with high pruning power. We distinguish between two types of variables during a run of a given CDCL SAT solver on a given formula: *glue variables* that appear in a *glue clause* up to the current state of the search and *non-glue variables* which do not appear in any of the glue clauses. We demonstrate that decisions with glue variables are more conflict efficient than non-glue variables. Based on this finding, we develop a variable bumping scheme named *glue bumping*, which lazily increases the activity score of glue variables to prioritize their selection. With benchmarks from the main track of SAT Competition 2017 and 2018, we demonstrate modest-to-strong performance gains with this scheme with four state-of-the-art CDCL SAT solvers. We also propose a novel conflict metric named G2L that measures the fraction of the learned clauses that are glue. To the best of our knowledge, for the first time this metric incorporates quality of the learned clauses in measuring conflict efficiency of CDCL solvers. We have also shown that G2L correlates well with the performance of CDCL SAT solvers, in cases where standard conflict metrics such as mean GLR and mean of average LBD scores do not provide consistent explanations.

- **Chapter 4**: In this chapter, we have introduced two novel concepts for CDCL SAT search: a Conflict Depression (CD) phase, in which the search does not produce any conflicts for a number of consecutive decisions and a Conflict Burst (CB) phase, in which the search produces at least one conflict in each of a number of consecutive decisions. With 750 instances from SAT Competition 2017 and 2018 and two high-performance CDCL SAT solvers, we demonstrated empirically that the typical search behavior in CDCL consists of shorter but conflict intense CB phases, followed by longer CD phases.

  We showed that there is a weak correlation between average length of CD phases and solving hardness. We studied the correlation between backjumps and CD. Our analysis shows that in general the correlation is low for the majority of instances. However, on average, there is a high correlation between the length of a CD phase and length of backjumpings for a majority of instances.

- **Chapter 5**: To escape from CD phases quickly, we propose an algorithmic extension of the standard CDCL framework named `expSAT` that performs random exploration amid substantially large CD phases. The goal of performing exploration is to identify, and then prioritize conflict friendly variables, selection of which may generate a conflict quickly and end the current CD phase. Our extensive evaluation of `expSAT` with 5 state-of-the-art CDCL SAT solvers show modest-to-strong performance gains on benchmarks from SAT Competition 2017 and 2018, and impressive gains with hard SATCoin instances. Analysis of the experimental data from two of our `expSAT` extensions and their corresponding baselines shows that these extensions reduce the average CD phase length for problem instances for which our `expSAT` extensions are more efficient.

- **Chapter 6:** A decision in CDCL can generate 0 or more conflicts. In this chapter, we study conflicts generating decisions in detail. First, we characterize single conflict (`sc`) decisions and multi-conflict (`mc`) decisions in terms of the quality of the learned clauses that each type produces, and show that `sc` decisions learn higher quality clauses than `mc` decisions, on average. Our theoretical analysis with `mc` decisions reveals an interesting insight: learned clauses in a `mc` decision are connected. This insight leads to the formulation of a new measure named `ConflictProximity`, that measures closeness between conflicts in a given sequence. With the concept of `ConflictProximity`, we demonstrate that on average conflicts occurring in a `mc` decision are more closely related than conflicts in a sequence of conflicts in `sc` decisions.

  We formulated `CRVR`, a decision strategy that lazily de-prioritizes the selection of variables which appear in some of the learned clauses. Our empirical evaluation of `CRVR` with benchmarks from SAT Competition 2020 show modest-to-strong performance gains with 3 leading CDCL SAT solvers for SAT instances.

7.2 Impact of this Thesis

We highlight the impact of this thesis work below:

- The conflict efficiency based characterization of glue variables is a concrete step towards the understanding of how CDCL branching heuristics work and why they work well. This aspect of Chapter 3 has been recognized in [25, 52].

  This characterization has opened up the possibility to design better CDCL SAT solving algorithms, such as the glue-bumping method. In a more recent work [32], the author has developed a machine learning model to predict glue-variables for making decisions, which helps in improving performance of the CDCL SAT solver CaDiCaL over recent SAT Competition benchmarks. We hope that further effective methods for CDCL SAT solving will be developed in the future following this characterization.

- The concept of conflict depression phases is new for CDCL SAT research. Besides exploration as in `expSAT`, a deeper understanding of conflict depression phases could lead to new research avenues for improving CDCL SAT search.

- The solvers that implement the techniques developed in this thesis have demonstrated strong performance in SAT Competition-2021 and SAT Race-2019. We expect that these strong performances will motivate SAT practitioners from both academia and industry to use our solvers for solving SAT problems.

- In SAT Competition-2018, we submitted a SAT solver named exp_MC, which is based on a preliminary version of the expSAT approach. Though exp_MC did not make it into the top tier of that competition, it was used in a recent work in Hierarchical Task Network (HTN) planning [10] domain, where exp_MC was the best performing system among 19 tested systems the authors have used for evaluating a novel SAT encoding of HTN planning. Thus exp_MC pushed the boundary for SAT based HTN planning.

- The process of bitcoin mining is very energy demanding and is a subject of increasing concern for the environmental effects of the mining industry [26, 75]. One effective solution could be making the mining process faster.

  As a proof-of-concept, the bitcoin mining process has been encoded into a

SAT benchmark named SATCoin [47], which opens up the possibility of using highly scalable CDCL solvers for mining bitcoins. The impressive performance gains with `expSAT` on the SATCoin benchmark is an indication that `expSAT` could be an useful technique in SAT based bitcoin mining.

7.3 Future Work

Here, we provide some research avenues that could be explored following this thesis work:

- **Chapter 3:**
  - The routine for updating heuristic scores in the glue bumping scheme uses a static rule. To further improve the performance of this scheme, devise a method that dynamically chooses a rule from a set of rules, depending on the current state of the search.
  - The *glue-level* measures the frequency of appearance of glue variables in glue clauses up to the current state of the search. A variable with high glue level logically connects many glue clauses in which it appears.

    In the Variable Incidence Graph (VIG) of a given SAT formula, the eigencentrality of a node measures the impact of that node in the VIG. Bridge variables in such a VIG connects two sub-graphs of that VIG. Are there any correlations between variables with high glue-levels with variables with high eigencentrality? What percentage of variables with high glue-levels are bridge variables?
  - A non-glue learned clause that contains a high number of glue variables could also be an important clause, and preventing the deletion of that clause could be beneficial for the search. Can we design clause deletion heuristics based on the notion of glue level?
  - The conflict efficiency metric G2L incorporates conflict quality in its measure. Can we design more efficient branching heuristics based on this measure of conflict efficiency?

- **Chapter 4**
  - The study of parameterized complexity of SAT attempts to explain solving hardness of SAT as a function of some structured parameters, such as backbones and backdoors [38], community structures [2, 53] and mergability [84]. None of those had been shown to be both theoretically and empirically satisfying in explaining hardness of SAT. Hence, relating solving hardness with a parameter is a very challenging problem in SAT research [28].

    Though in Chapter 4, we show a weak correlation between the phenomena of CD and solving hardness, the relationship between solving hardness and the CD phases is not well understood and remains an interesting future work.
  - While CD phases are pathological, for an UNSAT formula, the occurrence of some long CD phases may help the search learn a clause which is beneficial for constructing a refutation proof for the formula. How to verify this phenomena empirically is an interesting, challenging issue. Given a set of UNSAT formulas, one way is to analyze the Deletion Resolution Asymmetric Tautology (DRAT) [77] log of learned clauses for each formula and analyze how many of these learned clauses follow long CD phases.
  - Can we use machine learning models for better handling of CD phases? We see two research issues in this direction:
    * **<u>Issue I</u>**: In Chapter 4, we showed that for the majority of test instances, the correlation between the length of backjumps and the length of CD phases that follows those backjumps is weak-to-moderately positive (see Figure 4.12). In general, search gives no indication of how long a CD phase will last. A CDCL search cognisant of an impending long CD phase, could take *appropriate course of actions* (such as performing restart, resetting the variable selection/phase selection heuristics, reducing the current clause database)

just before the occurrence of that long CD phase. Two interesting research directions are: (i) develop machine learning models to predict the length of CD phases given the state of the search, and (ii) devise a CDCL algorithm that takes appropriate course of actions amid a CD phase, if the trained machine learned model predicts a long CD phase.

∗ **<u>Issue II</u>**: Develop (i) a machine learning model, which is trained to predict *conflict-friendly* branching variables *during a CD phase* and (ii) a CDCL algorithm that uses this model to guide the search to escape from a CD phase quickly.

- **Chapter 5:**
  - In general, performing deeper walks with an `expSAT` solver may incur high overhead than shallower walks. Are there SAT benchmarks for which deeper exploration can be helpful?
  - Lookahead based SAT solvers [11] guide the search by performing deterministic probing of the future states. Designing algorithms that perform deterministic probing amid a CD phase is another interesting research direction.
  - The performance of `expSAT` is quite impressive for the SATCoin benchmark.What characteristics of a domain (such as SATCoin) makes `expSAT` especially effective?
- **Chapter 6:**
  - Explanation the poor performance of the `CRVR` strategy on UNSAT instances, and then improve the performance of the CRVR method on UNSAT instances.
  - Some CDCL SAT sovlers such as CaDiCaL and Kissat use VMTF as their predominant decision heuristic along with VSIDS. Currently, `CRVR` is only applicable to activity based decision heuristics such as VSIDS

and LRB. How to apply the `CRVR` strategy to non-activity based selection heuristics such as VMTF.

- `CRVR` uses two fixed user defined parameters `Q` and `K`. How to design an adaptive strategy for these two parameters?
- In the `CRVR` method, we deprioritize the selection of a `poor` crv. Extending the current analytical/empirical results of this chapter and designing a method that prioritizes selection of a `crv` is another interesting research direction.

Bibliography

[1] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. "The Community Structure of SAT Formulas." In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. 2012, pp. 410–423. DOI: 10.1007/978-3-642-31612-8_31. URL: https://doi.org/10.1007/978-3-642-31612-8_31.

[2] Carlos Ansótegui et al. "Community Structure in Industrial SAT Instances." In: *J. Artif. Intell. Res.* 66 (2019), pp. 443–472.

[3] Albert Atserias et al. "Theoretical Foundations of Applied SAT Solving." In: *Workshop on Theoretical Foundations of Applied SAT Solving*. Banff International Research Station for Mathematical Innovation and Discovery. 2014, pp. 1–10.

[4] Gill Audemard and Laurent Simon. "Refining Restarts Strategies for SAT and UNSAT." In: *Proceedings of CP 2012*. 2012, pp. 118–126.

[5] Gille Audemard and Laurent Simon. "Extreme Cases in SAT Problems." In: *Proceedings of SAT 2016*. 2016, pp. 87–103.

[6] Gilles Audemard and Laurent Simon. *glucose 4.1 , https://www.labri.fr/perso/lsimon/glucose/ , accessed date: 2019-08-26.*

[7] Gilles Audemard and Laurent Simon. *glucose 4.2.1 , http://sat2018.forsyte.tuwien.ac.at/solvers/main_and_glucose_hack/, accessed date: 2019-08-26.*

[8] Gilles Audemard and Laurent Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers." In: *Proceedings of IJCAI 2009*, pp. 399–404.

[9] Gilles Audemard et al. "Boosting Local Search Thanks to CDCL." In: *Proceedings of LPAR 2010*. 2010, pp. 474–488.

[10] Gregor Behnke, Daniel Höller, and Susanne Biundo. "Bringing Order to Chaos - A Compact Representation of Partial Order in SAT-based HTN Planning." In: *AAAI 2019*. 2019, pp. 7520–7529.

[11] A. Biere et al. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands: IOS Press, 2009. ISBN: 1586039296, 9781586039295.

[12] Armin Biere. “CaDiCaL at the SAT Race 2019.” In: *Proceedings of SAT Race 2019*. 2019, pp. 8–9.

[13] Armin Biere et al. “CaDiCaL, Kissat, Paracooba, PLingeling and Treengeling Entering the SAT Competition 2020.” In: *Proceedings of SAT Competition 2020*, pp. 50–53.

[14] Cameron Browne et al. “A Survey of Monte Carlo Tree Search methods.” In: *IEEE Trans. Comput. Intellig. and AI in Games* 4.1 (2012), pp. 1–43.

[15] Cristian Cadar et al. “EXE: automatically generating inputs of death.” In: *Proceedings of CCS 2006*. 2006, pp. 322–335.

[16] Shaowei Cai and Xindi Zhang. “Deep Cooperation of CDCL and Local Search for SAT.” In: *SAT-2021*, pp. 64–81.

[17] Md. Solimul Chowdhury, Martin Müller, and Jia You. “Guiding CDCL SAT Search via Random Exploration amid Conflict Depression.” In: *Proceedings of AAAI 2020*, pp. 1428–1435.

[18] Md Solimul Chowdhury, Martin Müller, and Jia-Huai You. “Exploiting Glue Clauses to Design Effective CDCL Branching Heuristics.” In: *Proceedings of CP-2019*, pp. 126–143.

[19] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures.” In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. 1971, pp. 151–158.

[20] Martin Davis, George Logemann, and Donald W. Loveland. “A machine program for theorem-proving.” In: *Commun. ACM* 5.7 (1962), pp. 394–397.

[21] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory.” In: *J. ACM* 7.3 (1960), pp. 201–215.

[22] Niklas Eén and Armin Biere. “Effective Preprocessing in SAT Through Variable and Clause Elimination.” In: *Proceedings of SAT 2005*. 2005, pp. 61–75.

[23] Niklas Eén and Niklas Sörensson. “An Extensible SAT solver.” In: *Proceedings of SAT 2003. Selected Revised Papers*. 2003, pp. 502–518.

[24] Michael Färber, Cezary Kaliszyk, and Josef Urban. “Monte Carlo Tableau Proof Search.” In: *Proceedings of CADE 2017*. 2017, pp. 563–579.

[25] Mathias Fleury. “Formalization of Logical Calculi in Isabelle/HOL.” PhD thesis. Universität des Saarlandes Saarbrücken, 2020.

[26] Spyros Foteinis. “Bitcoin’s alarming carbon footprint.” In: *Nature* 554, 169 (2018), pp. 1–15.

[27] Jeremy Frank, Peter C. Cheeseman, and John C. Stutz. “When Gravity Fails: Local Search Topology.” In: *J. Artif. Intell. Res.* 7 (1997), pp. 249–281.

[28] Vijay Ganesh and Moshe Y. Vardi. “On the Unreasonable Effectiveness of SAT Solvers.” In: *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2020, pp. 547–566.

[29] Carla P. Gomes et al. “Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems.” In: *J. Autom. Reasoning* 24.1/2 (2000), pp. 67–100.

[30] Aarti Gupta, Malay K. Ganai, and Chao Wang. “SAT-Based Verification Methods and Applications in Hardware Verification.” In: *Proceedings of SFM 2006*. 2006, pp. 108–143.

[31] Shai Haim and Marijn Heule. “Towards Ultra Rapid Restarts.” In: *CoRR* abs/1402.4413 (2014).

[32] Jesse Michael Han. “Enhancing SAT solvers with glue variable predictions.” In: *CoRR* abs/2007.02559 (2020).

[33] Jörg Hoffmann. “Where Ignoring Delete Lists Works: Local Search Topology in Planning Benchmarks.” In: *J. Artif. Intell. Res.* 24 (2005), pp. 685–758.

[34] Matti Järvisalo, Armin Biere, and Marijn Heule. “Blocked Clause Elimination.” In: *Proceedings of TACAS 2010*. 2010, pp. 129–144.

[35] Matti Järvisalo, Marijn Heule, and Armin Biere. “Inprocessing Rules.” In: *Proceedings of IJCAR 2012*, pp. 355–370.

[36] James L. Johnson. *Probability and Statistics for Computer Science*. Wiley-Interscience, 2008. ISBN: 9780470383421.

[37] George Katsirelos and Laurent Simon. “Eigenvector Centrality in Industrial SAT Instances.” In: *Proceedings of CP 2012*. 2012, pp. 348–356.

[38] Philip Kilby et al. “Backbones and Backdoors in Satisfiability.” In: *Proceedings of AAAI-2005*, pp. 1368–1373.

[39] Steven M LaValle. *Planning algorithms*. Cambridge University Press, 2006.

[40] Chu Min Li and Anbulagan. “Look-Ahead Versus Look-Back for Satisfiability Problems.” In: *Proceedings of CP 1997*, pp. 341–355.

[41] Jia Hui Liang et al. “An Empirical Study of Branching Heuristics Through the Lens of Global Learning Rate.” In: *Proceedings of SAT 2017*, pp. 119–135.

[42] Jia Hui Liang et al. “Exponential Recency Weighted Average Branching Heuristic for SAT Solvers.” In: *Proceedings of AAAI 2016*, pp. 3434–3440.

[43] Jia Hui Liang et al. “Learning Rate Based Branching Heuristic for SAT Solvers.” In: *Proceedings of SAT 2016*, pp. 123–140.

[44] Jia Hui Liang et al. *MapleSAT: Combining Machine Learning and Deduction in SAT solvers* , `https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/,accesseddate:2019-08-26`.

[45] Jia Hui Liang et al. “Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers.” In: *Proceedings of Hardware and Software: Verification and Testing 2015*, pp. 225–241.

[46] Mao Luo et al. “An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers.” In: *Proceedings of IJCAI 2017*, pp. 703–711.

[47] Norbert Manthey and Jonathan Heusser. “SATcoin - Bitcoin mining via SAT.” In: *Proceedings of SAT Competition 2018*. 2018, pp. 67–68.

[48] Fabio Massacci and Laura Marraro. “Logical Cryptanalysis as a SAT Problem.” In: *J. Autom. Reasoning* 24.1/2 (2000), pp. 165–203.

[49] Matthew W. Moskewicz et al. “Chaff: Engineering an Efficient SAT Solver.” In: *Proceedings of DAC 2001*, pp. 530–535.

[50] Alexander Nadel and Vadim Ryvchin. “Chronological Backtracking.” In: *Proceedings of SAT 2018*, pp. 111–121.

[51] Hootan Nakhost and Martin Müller. “Monte-Carlo Exploration for Deterministic Planning.” In: *Proceedings of IJCAI 2009*, pp. 1766–1771.

[52] Adri‘a Navarro. “Understanding Literal Block Distance in SAT solvers.” Bachelor Thesis, University of Barcelona, 2020.

[53] Zack Newsham et al. “Impact of Community Structure on SAT Solver Performance.” In: *Proceedings of SAT-2014*, pp. 252–268.

[54] Chanseo Oh. “Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL.” PhD thesis. New York University, 2016.

[55] Chanseok Oh. “Between SAT and UNSAT: The Fundamental Difference in CDCL SAT.” In: *Proceedings of SAT 2015*. 2015, pp. 307–323.

[56] K Pipatsrisawat and Adnan Darwiche. “A Lightweight Component Caching Scheme for Satisfiability Solvers.” In: *Proceedings of SAT 2007*, pp. 294–299.

[57] Knot Pipatsrisawat and Adnan Darwiche. “On the Power of Clause-Learning SAT Solvers with Restarts.” In: *Proceedings of CP 2009*, pp. 654–668.

[58] Alessandro Previti et al. “Monte-Carlo Style UCT Search for Boolean Satisfiability.” In: *Proceedings of AI*IA 2011*, pp. 177–188.

[59] Bruce Ratner. “The correlation coefficient: its values range between +1/1, or do they?” In: *Journal of Targeting, Measurement and Analysis for Marketing* 17.1/2 (2000), pp. 139–142.

[60] Jussi Rintanen. “Engineering Efficient Planners with SAT.” In: *Proceedings of ECAI 2012*. 2012, pp. 684–689.

[61] Christopher D. Rosin. “Nested Rollout Policy Adaptation for Monte Carlo Tree Search.” In: *Proceedings of IJCAI 2011*, pp. 649–654.

[62] Lawrence Ryan. “Efficient algorithms for clause-learning SAT solvers.” MA thesis. Simon Fraser University, 2004.

[63] *SAT Competition 2017 Benchmarks,* *`https://baldur.iti.kit.edu/sat-competition-2017/index.php?cat=downloads`*, *accessed date: 2021-03-10.*

[64] *SAT Competition 2017 Solvers,* `https://baldur.iti.kit.edu/sat-competition-2017/solvers/`*, accessed date: 2019-04-25.*

[65] *SAT Competition 2018 ,* `https://www.labri.fr/perso/lsimon/glucose/`*, accessed date: 2019-08-26.*

[66] *SAT Competition 2018 Benchmarks,* `http://sat2018.forsyte.tuwien.ac.at/benchmarks/`*, accessed date: 2021-03-10.*

[67] *SAT Competition 2020,* `https://satcompetition.github.io/2020/downloads.html`*, accessed date: 2021-03-06.*

[68] *SAT Competition 2021 ,* `https://satcompetition.github.io/2021/` *(2021-08-26).*

[69] *SAT Race 2019,* `http://sat-race-2019.ciirc.cvut.cz/solvers/`*, accessed date: 2021-03-10.*

[70] Bart Selman, Henry A. Kautz, and Bram Cohen. "Local search strategies for satisfiability testing." In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop 1993*. 1993, pp. 521–532.

[71] Bart Selman, Hector J. Levesque, and David G. Mitchell. "A New Method for Solving Hard Satisfiability Problems." In: *Proceedings of AAAI 1992*, pp. 440–446.

[72] João P. Marques Silva and Karem A. Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability." In: *IEEE Trans. Computers* 48.5 (1999), pp. 506–521.

[73] David Silver et al. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587 (2016), pp. 484–489.

[74] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems." In: *Proceedings of SAT 2009*, pp. 244–257.

[75] Christian Stoll, Lena Klaaßen, and Ulrich Gallersdörfer. "The carbon footprint of bitcoin." In: *Joule* (2020), pp. 1–15.

[76] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 2nd Edition. Cambridge, MA, USA: MIT Press, 2018. ISBN: 0262193981.

[77] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. "DRAT–trim: Efficient Checking and Trimming Using Expressive Clausal Proofs." In: *Proceedings of CP-2014*, pp. 422–429.

[78] Step Wolfram. *Rule 30 in Elementary Cellular Automaton,* `https://mathworld.wolfram.com/Rule30.html`*, accessed date: 2020-04-09.*

[79] Stephen Wolfram. *A new kind of science*. Wolfram-Media, 2002. ISBN: 978-1-57955-008-0.

[80] Fan Xiao et al. "MapleLRB_LCM, Maple_LCM, Maple_LCM_Dist, MapleLRB_LCMoccRestart and Glucose3.0+width in SAT Competition 2017." In: *Proceedings of SAT Competition 2017*, pp. 22–23.

[81] Fan Xie, Hootan Nakhost, and Martin Müller. “Planning Via Random Walk-Driven Local Search.” In: *Proceedings of ICAPS 2012*, pp. 315–322.

[82] Fan Xie et al. “Type-Based Exploration with Multiple Search Queues for Satisficing Planning.” In: *Proceedings of AAAI 2014*, pp. 2395–2402.

[83] Neng-Fa Zhou. *Picat,* `http://picat-lang.org/resources.html`*, accessed date: 2020-04-09.*

[84] Edward Zulkoski et al. “The Effect of Structural Measures and Merges on SAT Solver Performance.” In: *Proceedings of CP-2018*, pp. 436–452.

Appendix A

Safe Population Growth with Rule-30: A SAT Benchmark

This benchmark was developed for SAT Competition 2021 [68].

A.1 Introduction

A *population* is an one-dimensional grid of $n \geq 1$ organisms, where each organism evolves between being alive (1) and dead (0) over chronological time steps by following a fixed rule of evolution. At any time step $t \geq 1$, the combined states of n organisms represent the state of the population at t. At t, a population is under the *threat of extinction*, if the number of alive organisms falls below $n * (P/100)$, where $0 < P \leq 100$, and *safe* otherwise. We say that a population *grows* over T time steps, if for any time step $t < T - 1$, population at $t + 1$ has more alive organisms than population at t.

In our proposed SAT benchmark *Safe Population Growth (SPG)*, given a population of n organisms, and a maximum time step T, verify if a population could *safely grow* up to time step T, while following a fixed rule of evolution. 20 instances of this benchmark were submitted for the SAT Competition 2021.

A.2 SPG as a Cellular Automaton

State evolution in the Safe Population Growth (SPG) problem deals with cells in finite elementary cellular automaton (CA) [79], with respect to (i) the *safety* constraint at any given time step and (ii) the *growth* constraint between any two consecutive time steps.

In an elementary CA, at time step $t+1$, the state of a cell c, which has cell l (resp. r) as its left (resp. right) neighbour, is computed based on a boolean combination of the states of c, l, and r at time t. There are $2^3 = 8$ combinations of boolean values for l, c, and r at t, for each of which there are 2 ways to set the value of the state of c at $t + 1$. Hence, there are $2^8 = 256$ ways to set this new state, which are called *rules* [79] for a given elementary CA.

These rules are divided into four classes based on their patterns of evolution in a given CA. We consider *Rule 30* [78] for the SPG problem, an instance of a class IV, which is known to exhibit chaotic behavior. At time $t+1$, for a given center cell

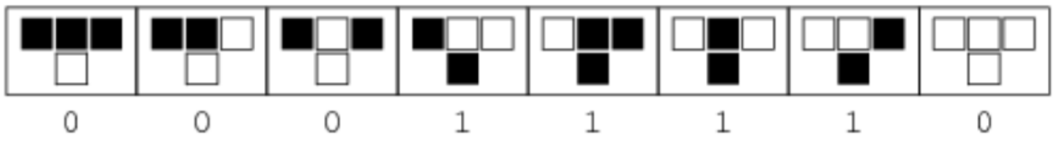


Figure A.1: State evolution for the center cell for Rule 30; black cells represent alive (1) cells, white cells represent dead (0) state.

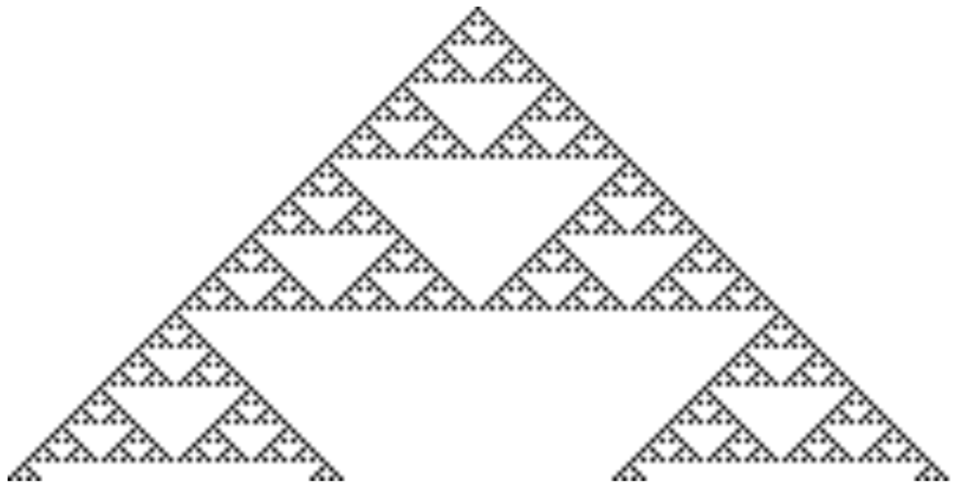


Figure A.2: Emergence of fractal patterns with Rule 90 that applies the rule $c^{t+1} \leftarrow (l^t \ \mathbf{XOR} \ r^t)$, for any time step $t \geq 1$.

(*c*), its left (*l*) and right (*r*) neighbours, Rule 30 computes the state $center^{t+1}$ of the *center* cell as follows:

$$c^{t+1} \leftarrow l^t \ \mathbf{XOR} \ (c^t \ \mathbf{OR} \ r^t)$$

Figure A.3 (also taken from [78]) shows a chaotic evolution of a CA that follows Rule 30, where the evolution of the cellular automaton does not exhibit any regularity. Figure A.2 shows a contrasting example with Rule 90 from class III, where a cellular automaton develops clear fractal patterns.

A.3 SAT encoding of the SPG problem

A.3.1 SPG as a SAT Benchmark

Given a population of $n \geq 1$ organisms, a maximum time step $T \geq 2$, , and a safety threshold $0 < P \leq 100$, the task of the SPG problem is to determine if the population evolves up to T by following Rule 30, with respect to the following two constraints:
Safety: *The Total number of alive organisms in every time step* $1 \leq t \leq T$ *is at least* $n * (P/100)$.

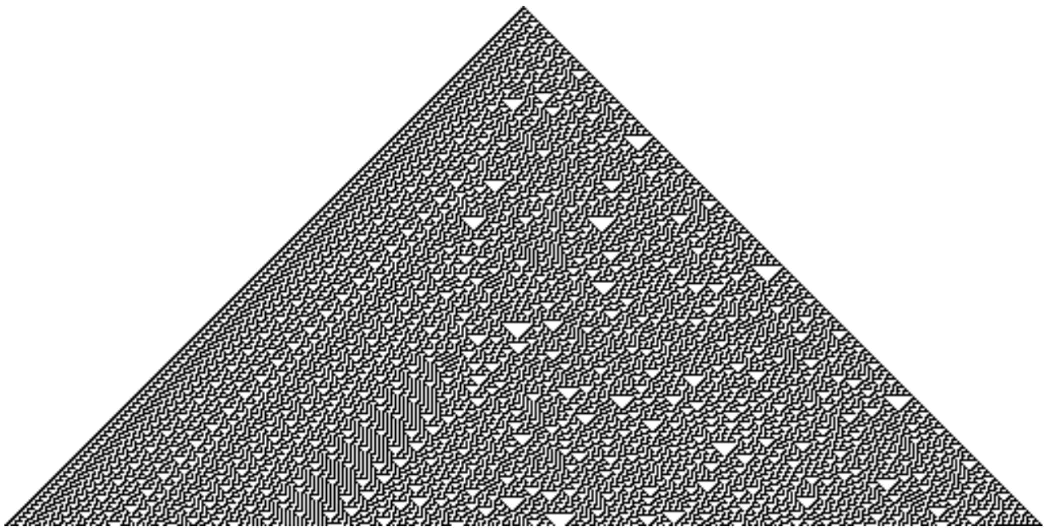


Figure A.3: Emergence of chaotic behavior with Rule 30; an application of Rule 30 produces extremely irregular and complex patterns.

Growth: *For any two consecutive time steps, the number of alive cells at* $t+1$ *is greater or equal to the number of alive cells at* t.

We can encode an instance of the SPG problem as a SAT instance. Let s_i^t be the state of the current cell i, where $1 \leq t \leq T$ and $1 \leq i \leq n$. Given a SPG problem, we encode it as a SAT formula F_{SPG} as follows

$$F_{SPG} = F_{evolution} \cup F_{safety} \cup F_{growth} \cup F_{boundary}$$

where $F_{evolution}, F_{safety}, F_{growth}$, and $F_{boundary}$ are defined as follows:

$$F_{evolution} : \bigwedge_{t=1}^{T} \bigwedge_{i=1}^{n} (s_i^{t+1} = (s_{i-1}^t \oplus (s_i^t \vee s_{i+1}^t)))$$

$$F_{safety} : \bigwedge_{t=1}^{T} \sum_{i=1}^{n} s_i^t \geq n * (P/100)$$

$$F_{growth} : \bigwedge_{t=1}^{T-1} \sum_{i=1}^{n} s_i^{t+1} \geq \sum_{i=1}^{n} s_i^t$$

$$F_{boundary} : \bigwedge_{t=1}^{T} \neg s_0^t \wedge \neg s_{n+1}^t$$

Over T time steps,

- $F_{evolution}$ encodes the evolution of the population of n organisms that follows Rule 30.
- F_{safety} encodes the population safety constraint.
- F_{growth} encodes the population growth constraint.

- $F_{boundary}$ encodes the assertion that the left (resp. right) neighbor of the left-most (resp. rightmost) organism outside of the *boundary* of a given population is always dead (0).

F_{SPG} is SATISFIABLE, if the population can evolve up to time step T with respect to the safety and growth constraint, otherwise, it is UNSATISFIABLE.

A.4 Problem Modeling and Instance Generation for the SPG benchmarks

A.4.1 Problem Modeling

picat [83] is a CSP solver which accepts a CSP problem and converts it to a SAT CNF formula, which is in turn solved by a SAT solver hosted by **picat**. Before solving the converted CNF formula, **picat** outputs the CNF formula.

To generate instances for the SPG benchmark, we first model the SPG problem in a **picat** program $picat_{SPG}$. Then, for a given set of parameter values for (T, n, P), we generate the CNF F_{SPG} from $picat_{SPG}$ by using the CNF generation functions of **picat**.

Appendix B

Source Code Links

B.1 Chapter 3

- Four CDCL solvers that implement the Glue Bumping method are available at: `https://github.com/solimul/gluebumping`

B.2 Chapter 5

- Eleven CDCL solvers that implement the `expSAT` method are available at: `https://github.com/solimul/expSATExtensions`

B.3 Chapter 6

- Three extensions of the `CRVR` method are available at: `https://figshare.com/articles/software/CRVR_Extensions/14229065`