Department of Creative Informatics

Graduate School of Information Science and Technology

THE UNIVERSITY OF TOKYO

Doctoral Thesis

Widely Applicable Approaches to Adjust Intensity and Diversity for Constructing Efficient SAT Solvers

（集中性と多様性を調整し SAT ソルバー性能向上を目指す

広く適用可能なアプローチ）

Seongsoo Moon

文 性琇

Supervisor: Professor Mary Inaba

Abstract

The satisfiability (SAT) problem is a well-known NP-complete problem. Generally, there are no polynomial-time solutions for NP-complete problems. In the last two decades, substantial progress has been made on SAT algorithms; now, many application problem instances from domains such as puzzles, circuit verification, and planning can be encoded easily into SAT problems and solved using SAT solvers. Several elements, such as conflict-driven clause learning (CDCL), restarts, propagation using a lazy data structure, and conflict-based branching heuristics, have played a significant role in the dramatic progress of SAT solvers. Furthermore, throughout the evolution of multicore hardware, many types of parallel SAT solvers have been proposed. Combinatorial search with a parallel SAT solver shares information among workers to diversify the search and avoid duplicate work. However, when the number of workers or the size of a problem increases, the amount of information increases exponentially. When the amount of information increases, the cost of maintaining and utilizing information also increases. This can cause a problem of search efficiency for massively parallel environments. In recent years, a portfolio-based approach has become mainstream for parallel SAT solvers. In portfolio approaches, maintaining the diversification and intensification tradeoff is very important.

In this paper, we propose several methods for achieving efficient SAT solvers by adjusting search intensity and diversity. First, we propose breaking ties in branching heuristics. Branching heuristics decide which variable to branch on next during a tree search. Many of them have been proposed for search intensification of SAT solvers. We recognize the existence of ties inherent in branching heuristics and propose a method for breaking ties to enhance search intensification. Our approach is designed to intensify the interplay between branching heuristics and clause learning schemes. Second, we propose a hybrid model to secure search diversity and integrate different algorithms for building efficient SAT solvers. Our objective here is to provide an efficient single solver that can be reused for other research as a base solver. As a first step toward this, we applied a random forest model to integrate several branching heuristics. This model works as preprocessing to select an adequate branching heuristic for each SAT instance. All branching heuristics in our model use the same data structure. Hence, they can be implemented easily in a single solver. Finally, we propose an approximate history map (AHM) to share information among workers in a parallel SAT solver using only a small amount of memory. The AHM concept can be applied to a multitude of scenarios to manage search intensity and diversity. This map is applicable in massively parallel environments at low cost. To achieve an AHM in parallel SAT solvers, we propose a Polarity Search Space Index (*PSSI*). After the construction of an AHM with a *PSSI*, we propose a sparsely visited area walking on search space (SaSS) heuristic as an application of the AHM. All of our proposals are evaluated through the benchmarks from SAT Competitions.

概要

充足可能性 (SAT、Satisfiability) 問題は NP 完全であると知られており、一般的に多項式で解を求める方法は知られていない。ここ 20 年に渡って、SAT 問題を解くアルゴリズムは飛躍的に発達しており、パズル、回路検証、プランニングなど様々なアプリケーション問題を SAT 問題に変換し、SAT solver を用いて容易に解決出来るようになって来た。これに貢献した要素技術としては、*conflict-driven clause learning* (CDCL)、リスタート、lazy data structure を用いた propagation、conflict ベースの branching heuristic などが挙げられる。さらに、multicore など計算資源の発達により並列 SAT solver も数多く提案されて来た。並列 solver で combinatorial search をする時は search の多様性確保や中腹探索を避けるために情報の共有を行う。しかし、ワーカーの数が増える、または問題のサイズが増加すると、情報量は指数的に増えて行く。これは超並列環境での search の効率に問題を発生させる原因と成り得る。近年、並列 solver の主流は portfolio-based approach が良く用いられている。portfolio では search の diversification と intensification をバランス良くハンドルすることが大事であると言える。

本研究では、集中性と多様性に着目し、その調整により効率的なソルバーを構築するためのいくつかの提案を行う。１つ目は、branching heuristics のための tie break である。Branching heuristics は SAT ソルバーの tree search 中にどの変数を選択するかを決める heuristics であり、多くのものは探索のソルバー集中性高めるために提案されている。我々は branching heuristics に内在している tie の頻繁な発生に着目し、さらに探索の集中性を高めるための tie break の手法を提案する。このアプローチは branching heuristics と clause learning scheme の相互作用を考慮して考案している。2 つ目に、ソルバーの多様性を確保し、複数のアルゴリズムを統合してソルバーの性能向上をするための hybrid model を提案を行う。我々の目標は効率的かつ他の研究でベースソルバーとして用いられるような single ソルバーを提供することである。その第一ステップとして、branching heuristics を統合するために random forest model を適用する。得られたモデルはソルバーに問題が与えられたら適切な branching heuristic を選択してくれる preprocessing の役割をする。モデル学習に用いられている branching heuristics すべては共通のデータ構造で実現出来るため簡単に single ソルバーで纏めることが出来る。最後に、並列ソルバーのワーカ間でメモリを節約しながら情報共有を行うために approximate history map (AHM) を提案する。AHM の概念はソルバーの探索の集中性と多様性を調整する様々なシナリオーに適用可能なものであり、超並列環境に置

いても低コストで用いることが出来ると期待される。AHM を SAT ソルバーの中で構築するために Polarity Search Space Index (*PSSI*) を提案する。*PSSI* をベースに AHM を構築し、その１つの応用例として sparsely visited area walking on search space (SaSS) の提案を行う。それぞれの提案は SAT Competitions のベンチマークを通して評価を行う。

Acknowledgements

I would like to thank my supervisor, Mary Inaba, and advisor, Hiroshi Tezuka, for their sincere advice and support throughout my PhD years. Professor Mary Inaba taught me many things: what is important for PhD research, how to write a paper, and how to express my idea to people. She also did financial supports: travel support for international conferences, research assistant, and experimental environments for the evaluation of my works. Researcher Hiroshi Tezuka provided technical support: how to manage linux server, how to manage source code, and troubleshooting for linux server. I would also like to thank committee members of my thesis for their time and advice on this thesis: Masayuki Inaba, Kenji Yamanishi, Shigeru Chiba, Toshiya Hachisuka, and Hideki Nakayama. I would also like to extend my heartfelt gratitude to my family and friends for their continued concern and love. Lastly, I would like to thank the Korean Scholarship Foundation. Owing to their finantial supports, I have been able to concentrate on my research.

Contents

Chapter 1

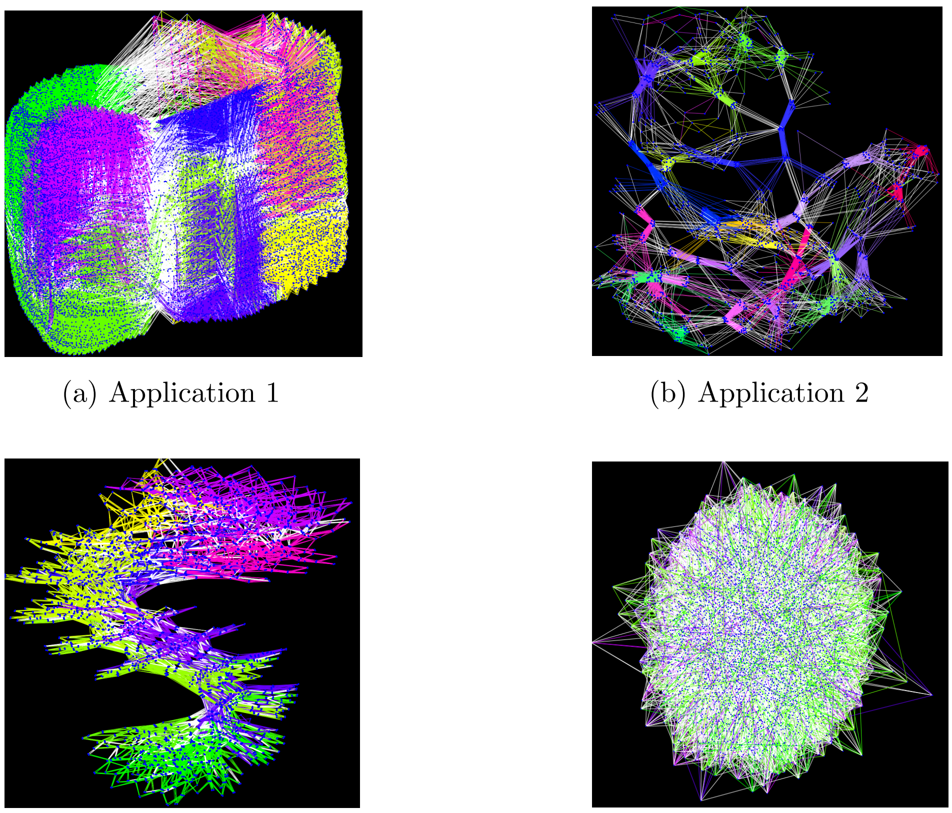
Introduction

In this chapter, we briefly describe SAT problems and their solvers and provide an overview of our research.

1.1 Satisfiability

The satisfiability (SAT) problem, determining whether there exists any assignment of variables that satisfies a given Boolean formula, is one of the most fundamental problems in computer science. This problem was firstly proved to be an NP-complete problem [1], thus all of the problems in the NP-complete class can be converted into the SAT problems within a polynomial time. It is widely believed that there is no algorithm for solving an NP-complete problem in a polynomial time, even though several theorists (including Donald E. Knuth) believe that P = NP [2].

SAT formulas can be divided into three categories, random, application and crafted formulas depending on their generations. Studies of random formulas aim to understand the hardness of formulas, such as finding a satisfiability threshold based on rigorous mathematics [3][4]. Random 3-SAT problems are the most representative. All of the clauses in a 3-SAT problem contain three variables. This is important, because all SAT problems are reducible to 3-SAT problems. Application problems are generated from industrial domains, such as planning [5], software and hardware verification [6], Bounded Model Checking [7] and circuit verification [8]. Many researchers believe that application problems have biased hidden structures and devise practical algorithms differently from those for random problems. Crafted problems are designed to be difficult to solve. Problems in



(a) Application 1

(b) Application 2

(c) Crafted

(d) Random

Fig. 1.1: Visualizations of community structures for several formulas from application, crafted, and random categories using SatGraf.

this category are more challenging than those came from application areas, but efficient algorithms for application problems can also be applied to solve these problems.

State-of-the-art SAT solvers have improved remarkably especially for application problems, and now many instances with hundreds of thousands of variables and millions of clauses can be solved using SAT solvers. Figure 1.1 shows the internal structures of several formulas came from different areas. These structures are generated using SatGraf [9], which visualizes a SAT formula based on its communities through community detection algorithms such as the Louvain method [10]. Each blue point in Figure 1.1 corresponds to a variable in a SAT formula. There is an edge between two variables, if they are included in the same clause. A distinct color is assigned for each community, and inter-community edges within the same community have the same color. Intra-community edges are white. Both application and crafted problems appear to have community structures in contrast

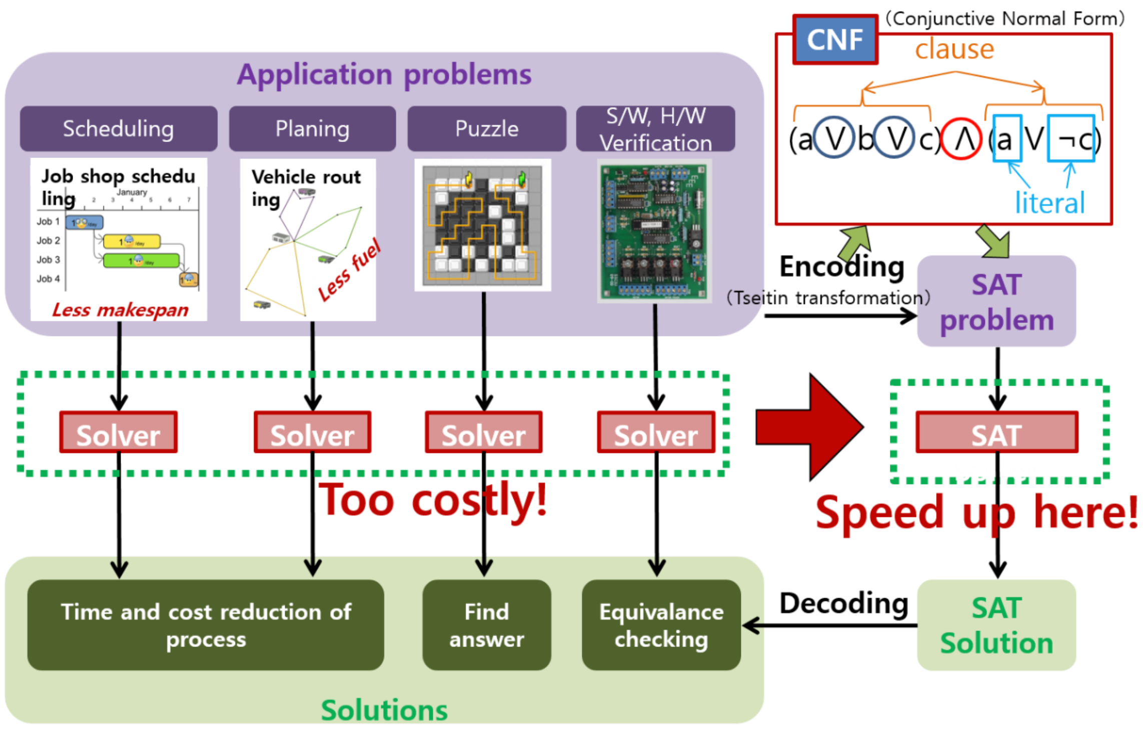


Fig. 1.2: Importance of SAT solver - universal solver for application problems

with random problems.

Figure 1.2 illustrates the importance of SAT solvers for application problems. As shown in the figure, we can create and optimize a solver for each problem, but that requires substantial resources. The development of a universal solver is a necessary, and researchers from many application domains are concentrating on improving the performance of SAT solvers. The encoding and decoding portions in Figure 1.2 can be performed in polynomial time, and thus increasing the speed of a SAT solver is directly connected to the solution times for many application problems. Problems must be converted into conjunctive normal form (CNF) to be solved using SAT solvers. Further information will be provided in Chapter 2.

1.2 Research overview

In this paper, we discuss several approaches to the construction of efficient SAT solvers for the industrial problems with consideration of diversification and intensification. We first provide a general overview of the diversification and intensification in SAT research, and then provide an overview of our research.

1.2.1 Diversification and intensification

The SAT problem is NP-complete. Therefore, simply traversing a tree to find a solution requires expenential time, in general. However, if there exists a particular structure in a SAT problem, we might find a solution efficiently using methods such as pruning or prioritizing the areas in a search space. In contrast to the random instances, many application instances are believed having hidden structures, as shown in Figure 1.1. Therefore, many researchers are working to adjust the intensification and diversification of SAT solvers adequately for searching efficiently.

When we begin solving an application problem, we do not know where a solution exists. We cannot quantify the structure of a problem or find a relationship between a structure and a distribution of solutions. However, we believe there exist hidden structures in a problem, and that well-designed algorithms might find a solution efficiently using those hidden structures. Consider a solver that is searching for a solution under some restrictions. A solver can search a solution more intensively if we add more restrictions. If we remove some restrictions, a solver has opportunities to search at different areas. We must determine whether to add or remove restrictions based on the current search status.

Let us say that we have an accurate measure showing the possibility of the existence of a solution in a subspace. If the measure shows that there is no solution under the current restrictions, a solver can move to a different subspace by removing and adding restrictions. We can also make a solver move to a different subspace when a measure says the possibility of solution is too low in the current subspace. The possibility of finding a solution might become high, when we notice that there is no solution in most parts of the current subspace and then add a suitable restriction to remove these parts. If we have confidence that the current problem is unsatisfiable, then diversifying the search based on a measure might not be good policy. All subspaces have a possibility of zero, when a problem is unsatisfiable. For an unsatisfiability proof, a solver must obtain restrictions by intentionally traversing a subspace without a solution. However, the scenarios we mention here are difficult to deal with in that manner, because no accurate measure for subspaces exists.

Some scenarios require intensive search, and the others require search diversification. This is domain, instance, and subspace dependent. We must design a SAT solver to cover these scenarios efficiently as much as possible. Efforts to adjust diversification and intensification efficiently work in many instances from application domains. Application problems have biased structures, and these structures leave some subspaces without a solution and other subspaces with many solutions. SAT solvers find a solution efficiently using techniques such as prioritizing recently used variables for intensive search, restarting a search by removing all restrictions, random selection of subspaces, divide-and-conquer for a search space, configuration of portfolios for search diversification, and sharing of information among workers. Techniques for search intensification and diversification can be explained on a community structure of a SAT instance. A solver performs an intensive search by assigning values to variables in the same community recursively. For example in Figure 1.1 (a), a solver can perform an intensive search by focusing on the assignments to purple variables. Search diversification can be obtained by changing the order of communities for intensive search. For example in Figure 1.1 (a), a solver can perform an intensive search on a purple community first, move to a green community and perform intensive search in it, and then move to a blue community. If the solver cannot solve a problem using this order, it can change the order of communities, such as (green $\rightarrow$ purple $\rightarrow$ blue) or (blue $\rightarrow$ purple $\rightarrow$ green). Actually, adjusting intensification and diversification of a search on a community structure is more difficult than what we describe here. A community structure changes consistently through assignments, cancellations, and additionally learned restrictions.

We considered that there is substantial room for improvement of SAT solvers by adjusting the intensification and diversification. Each technique is designed for an intention to intensify or diversify search. We might strengthen the intentions by scrutinizing them. When we dynamically adjust the intensification and diversification tradeoff, a measure, indicator, or score is required for controls. Searches based on a measure might lead in incorrect directions from time to time, becuase the measure might be unsuitable or uninformative, thereby randomly generating ties and breaks. Other approaches involve applying or devising a method to obtain the intensity or diversity of SAT solvers. Each technique has its variants, and the efficiencies of many of them are different. Their integration into

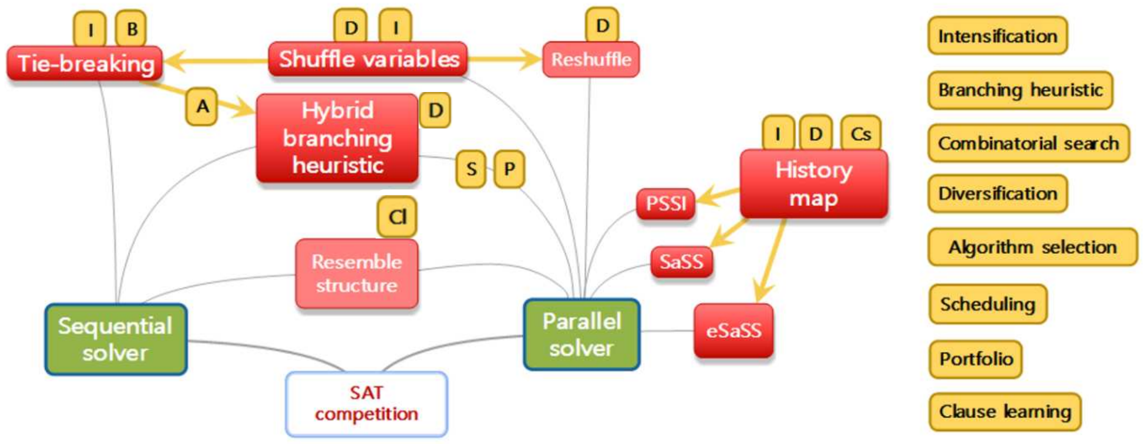


Fig. 1.3: Research overview - our proposals, their relations, and related keywords

a single solver might diversify a SAT solver. SAT solvers acquire information during a search, and retain it to avoid search redundancies. However, much of it is persistently erased, because a solver cannot retain it all. If we can record snapshots during a search by mapping and reducing the amount of information, then cumulative records might become useful for adjusting the diversification and intensification of SAT searches.

1.2.2 Our research

Each of our proposals is designed to adjust either intensification or diversification or both, and all of our approaches target a wide range of SAT solvers, not just specific solver. Our approaches can be applied directly to a variety of representative modern SAT solvers, and can also be used for ongoing research.

Figure 1.3 summarizes our research categories. The red boxes indicate our proposals, and the yellow boxes indicate related keywords. Each of the red boxes is connected to the sequential solver, the parallel solver, or both. We have several ideas for the light red boxes, but they are not yet complete or are not yet implemented. Solvers with our proposals can be submitted to the SAT Competition or evaluated through benchmarks from previous SAT Competitions. Each of the yellow boxes is positioned at a relevant point. Most of our proposals are related to at least one of diversification or intensification, indicating that our proposals are deeply related to these elements.

The boxes labeled tie-breaking, Hybrid branching heuristic, and History map indicate

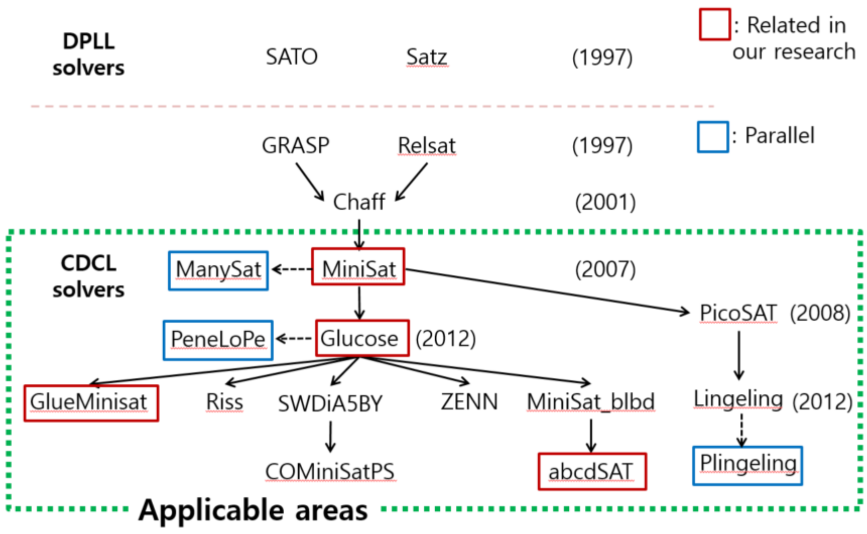


Fig. 1.4: Relations between our research and phylogenetic tree of representative SAT solvers

our principal proposals. We will discuss them shortly in the following sections. Details of each will be described in Chapters 4, 5, and 6, respectively. The box labeled Shuffle variables will be discussed in Chapter 2. We included that chapter, becuase the experiments in Chapter 2 encouraged us to proceed with our proposals.

We implemented several of our proposals in both sequential and parallel solvers and submitted them to SAT Competition 2016. We earned three medals, Best Crafted Benchmark Solver in the Main Track, and second prize and third prize in the Agile Track [11]. Detailed information regarding SAT Competitions will be provided in Section 2.2.

Figure 1.4 shows a phylogenetic tree of representative modern SAT solvers. Many modern solvers are generated from MiniSat and Glucose. Parallel solvers are generated from several kinds of sequential solvers, with the result that it is difficult to construct a tree using parallel solvers. All of our proposals can be applied widely within the rectangle with a green dotted line. Improvements in sequential solvers are related to improvements in parallel solvers, and a history map might be useful for solving a difficult instance with a sequential solver. In this paper, we applied the hybrid branching heuristic only for sequential solvers, but this model can be extended to parallel solvers when we consider a scheduling problem using several branching heuristics or simply select a portfolio.

1.3 New branching heuristic

Details about branching heuristics will be discussed in Section 2.5.

The branching heuristic is one of the most influential elements for improving the performance of SAT solvers. Several incomplete SAT solvers based on stochastic local search (SLS) do not require a branching heuristic. However, many complete SAT solvers use a branching heuristic to select the area in the search space. Especially, many SAT solvers with complete algorithms find a solution based on depth-first search (DFS). Davis-Putnam-Logemann-Lovveland (DPLL) [12] is one of the most prominent DFS algorithms with backtracking. DPLL SAT solvers select a variable to branch on next based on branching heuristics. Generally, branching heuristics are designed for intensive search by choosing a variable that became active recently.

Modern SAT solvers adopt conflict-driven clause learning (CDCL) [13] and are called CDCL solvers. The variable state independent decaying sum (VSIDS) [14] is the most representative branching heuristic for CDCL solvers and is widely used because of its efficiency has been demonstrated using benchmarks over the years. There have been many attempts to outcompete VSIDS, and several variants of VSIDS have been proposed. However, VSIDS is still widely used because of its robustness. Firstly, we attempted to improve VSIDS, because many modern SAT solvers are already using VSIDS as their branching heuristic. Therefore, the improvement of VSIDS is expected to connect to improved performance of many SAT solvers. We noted that ties occur in VSIDS and actually measured their frequencies. As far as we are aware, no one has actually measured tie occurrences in VSIDS or paid attention to breaking ties. We adopted the method of *tie-breaking*, because our intention was to improve VSIDS, not to outcompete it. VSIDS has already been proved its efficiency for a long time. If we propose a completely different branching heuristic, there would be a tradeoff. A new branching heuristic might help find a solution more rapidly for several instances, but also lose several instances as well.

The idea of breaking ties in branching heuristics was motivated from our preliminary experiments involving shuffling indexes of variables, which encouraged us to propose a hybrid branching heurisitic, as shown in Figure 1.3. During the shuffling experiments, we noticed that small changes in a branching heuristic had a substantial effect on the running time of solvers. Therefore, we thought that if we can affect small changes persistently, then the entire search efficiency would improve. Tie occurrences in branching heuristics are proper parts that enable our idea. Our *tie-breaking* attempted to give additional scores to variables that appeared to be more significant. We also considered the interplay among the branching heuristic and other techniques in a SAT solver. Our effort is to achieve more intensive search by applying *tie-breaking*. Details regarding our hybrid branching heuristic are provided in a following section.

We applied *tie-breaking* method primarily to VSIDS, and calling this *"tie-breaking of VSIDS"* (TBVSIDS). However, the *tie-breaking* concept can also be applied to many other branching heuristics. We additionally applied *tie-breaking* to the conflict history-based branching heuristic (CHB) [15], a recently proposed branching heuristic, and calling this the *"tie-breaking of CHB"* (TBCHB). Our sequential solver with our *tie-breaking* method was selected as the Best Crafted Benchmark Solver in the Main Track of SAT Competition 2016.

1.4 Hybrid branching heuristic

A SAT solver works as a universal solver to cover a variety of application categories. However, a single solver cannot cover all instances in general. Therefore, several algorithm selection studies [16][17] have been conducted to integrate different strategies in a single solver. A variety of SAT solvers have been proposed, as shown in Figure 1.4, and the differences among them enabled the learning of a model for algorithm selection. The models work differently based on their types, such as a preprocessing, a portfolio for parallel solvers, or a scheduling problem. The mainstream of these studies gathered N different state-of-the-art solvers and classified instances into N classes. Thus, a model selects a single SAT solver out of N SAT solvers on an instance basis. The performances of these multi-solver approaches are quite strong, but there is a major weakness in these approaches. They cannot serve as a base solver, such as Minisat or Glucose in Figure 1.4, because they already include several solvers. This means that they cannot be used for continuous research in spite of their outstanding performances.

Our objective here is to propose an efficient solver with a model of algorithm selection that could become a base solver for continuous research. To achieve this, we considered that algorithm selection must be performed in a confined space. SAT solvers contain many techniques consisting of many parts, and each solver chooses an algorithm and adjusts parameters for each part. Therefore, we considered that there are a large number of candidates for the selection of the confined space. We selected branching heuristics as a candidate in this paper, because we were proposing *tie-breaking* for branching heuristics. Therefore, we can observe the validity of *tie-breaking* for algorithm selection. Further, branching heuristics with *tie-breaking* do not require a new data structure, and can preserve the readability of a SAT solver as a base solver for ongoing research. We applied the random forest model for branching heuristics and proposed the hybrid branching heuristic, as shown in Figure 1.3. Currently, our hybrid model is designed only for sequential solvers choosing a branching heuristic as preprocessing. However, this model can be considered for constructing the portfolio for parallel SAT solvers or applied to the scheduling problem for dynamic switches of branching heuristics. We evaluate our hybrid model using different SAT solvers, because our intention is to provide a widely applicable hybrid model for a variety of SAT solvers.

1.5 History map

Parallel solvers share information for efficient search by handling the diversification and intensification of the search. However, when the number of workers or the size of a problem increases, the amount of information increases exponentially. It is difficult to extract the most valuable information to share from a large amount of information. Even if we extract important information successfully, it cannot be maintained indefinitely due to memory limitations and the degree of solver efficiency. Therefore, we were motivated to propose a scalable data structure for several reasons. First, the amount of shared information must be more limited, when the number of workers increases. Second, a large amount of information is erased periodically during the search. Third, we want to adjust the diversification and intensification of the search, minimizing the burden on time and resources.

To achieve these goals, we propose the approximate history map (AHM), wherein each snapshot captures approximate calculations for each worker, and in which the snapshots are accumulated to form a history map. The size of a map is fixed, and only a small space is required for this data architecture, regardless of the size of the problem or the number of workers. The AHM represents an approximate distribution of the number of visits to areas in the search space throughout the search. We predict which areas are sparsely (frequently) visited based on this map.

A precondition for achieving the AHM concept is that information must be extremely compressed. Our intuition is inspired by instruction prefetch research [18][19], because an extremely small space is allowed to record past memory accesses for the prediction of future memory accesses to improve efficiency. Access map pattern matching [19] uses a memory access map and can reduce past records successfully by dividing the memory address space into memory regions of a fixed size.

We propose using the AHM to handle diversification and intensification of a search. The AHM concept could be broadly applicable to solvers for combinatorial searching. We apply the AHM to a portfolio-based SAT solver in this paper, as shown in 1.3. We introduce a heuristic utilizing the proposed AHM, which we refer to as " sparsely visited area walking on search space " (SaSS). We show the effects of the SaSS heuristic experimentally.

1.6 Structure of this thesis

In this paper, we propose several methods for improving the performance of SAT solvers with consideration of diversification and intensification, as shown in Section 1.3.

Background for understanding the structures of modern SAT solvers is provided in Chapter 2. We explain the notion of SAT problem and provide a brief overview of SAT solvers especially for application problems. Then, we briefly explain several important elements that contribute substantially for improving the performance of SAT solvers. In Chapter 3, we discuss some preliminary experiments involving the index shuffling of variables. The contents of that chapter are introduced, because we obtained several intuitions and ideas from these experiments. Chapter 4 proposes a new branching heuristic with *tie-breaking*. The principal experimental results are related to TBVSIDS, and its performance is compared with those of VSIDS using SAT Competition benchmarks. In Chapter 5, we propose a hybrid branching heuristic as a first step of an algorithm selection research for SAT solvers. We introduce existing methods, address an issue, propose our idea for integration of SAT solvers, and explain the significance of our method. Chapter 6 proposes a history map. This concept can be expanded to general combinatorial search. We explain the concept of our history map and propose a method for constructing a map in a parallel SAT solver. As an example of AHM application, we propose a dynamic algorithm for diversifying search, and we evaluate it using SAT Competition benchmarks. Chapter 7 addresses several ideas that might be useful for improving SAT solvers, but that require further development or have yet to be implemented. In Chapter 8, we conclude and summarize several future work.

Chapter 2

Backgrounds

In this Chapter, we introduce fundamentals about SAT problems and SAT solvers. We show details about SAT problems first, explain basic search algorithm for SAT solvers and give overviews of several elements those are essential for modern SAT solvers.

2.1 Satisfiability problem

The Satisfiability (SAT) problem determines whether satisfying assignment exists for a given conjunctive normal form (CNF) formula. A CNF formula is a conjunction of clauses where a clause is consisted of a disjunction of literals. We say a clause is unary, binary, or ternary if it contains one, two, or three literals, respectively. Especially when a clause is unary, it is called unit clause. These clauses are considered important in general because they might be informative in many cases. A unit clause only contains a single variable x or $\neg x$ and forces $x = \mathit{TRUE}$ or $\neg x = \mathit{TRUE}$ ($x = \mathit{FALSE}$) to make itself satisfied. A binary or ternary clause can derive a unit clause with high probability through the chain of assignments. Each variable x corresponds to two literals: itself (x) and its negation ($\neg x$). Each literal can be assigned to either the value of *TRUE* or *FALSE*. Conversely, two different literals correspond to a variable and an assignment of a literal determines a value of its variable. Therefore Equation 2.1 satisfies for each variable x_n.

$$x_n = |\neg x_n| = |x_n| \tag{2.1}$$

Examples of SAT formulas are given in Equation 2.2 and 2.3.

$$\begin{aligned} F = (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4 \vee \neg x_1) \wedge (\neg x_4 \vee x_1 \vee \neg x_2) \\ \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (x_3 \vee x_4 \vee x_1) \end{aligned} \tag{2.2}$$

$$G = F \wedge (x_4 \vee \neg x_1 \vee x_2) \tag{2.3}$$

A formula F is satisfied with an assignment of $\{x_1, x_2, x_3, x_4\} = \{TRUE, FALSE, FALSE, FALSE\}$. However, when we construct a formula G by adding a clause shown in Equation 2.3, G is alway unsatisfied with any of assignments. Therfore, formulas F and G are evaluted as satisfiable and unsatisfiable. In general, each terms of satisfiable and unsatisfiable are simply expressed SAT and UNSAT respectively. However in this paper, we do not use these abbreviations to avoid misunderstandings. If we use these abbreviations and address, for example, “SAT instances”, this can be interpreted as either “Satisfiability instances” or “Satisfiable instances”. Therefore we write terms of satisfiable and unsatisfiable as it is for explicitness of the paper.

Examples of F and G might explain that distinguishing satisfiable problems from unsatisfiable problems is a very difficult task. Single additional clause changes a satisfiable formula F into a unsatisfiable formula G. Conversely, if we remove a clause randomly from G, then a formula changes into a satisfiable formula. A formula G is unsatisfiable because it meets either of Equation 2.4 and 2.5, and a clause removal from G makes a formula satisfiable because a removal makes these chains of implications be cut off.

$$x_1 \to \neg x_2 \to x_4 \to x_3 \to \neg x_1 \to x_2 \to \neg x_4 \to \neg x_3 \to x_1 \tag{2.4}$$

$$x_1 \to x_2 \to x_3 \to x_4 \to \neg x_1 \to \neg x_2 \to \neg x_3 \to \neg x_4 \to x_1 \tag{2.5}$$

In general, there exists satisfiability threshold for random SAT formula [4]. For example for random 3-SAT, instances get very hard to solve when a clauses-to-variables ratio in a formula is near 4.26, and when a ratio gets smaller or bigger, the probability of getting satisfiable or unsatisfiable rapidly gets higher, respectively. Thresholds such as 4.26 are not applied for application problems, because they contain biased hidden structures [20][21] in contrast to random problems. Therefore, SAT researches in the application areas have been improved in different ways compared with those of random areas. Our research in this paper are related to instances generated from application areas.

2.2 SAT solver

A SAT solver returns a solution when a SAT formula F have a solution and prove unsatisfiability if F is alway *FALSE*, i.e., $\bar{F}$ is tautology. A SAT problem is a well-known NP-complete problem and an unsatisfiability proof is a coNP-complete problem [22]. It is believed that there is no theoretically efficient algorithm for solving SAT problems, because the existence of a polynomial time algorithm would solve all the problems in a NP-complete class rapidly.

Even though SAT problems are included in a NP-complete class, instances came from application areas have biased structures, and modern SAT solvers can resolve many of these instances quickly within a reasonable timeframe as a result of researches from many different sorts of areas. Many researchers from several application areas are struggling to improve the performance of SAT solvers, because problems included in a NP-complete class can be reducible to SAT problems in polynomial time. Therefore, the improvement of SAT solvers would benefit to a lot of application areas.

A propositional formula can be efficiently converted into a CNF formula through Tseitin transformations [23]. For example, if we transform disjunctive normal form (DNF) (Equation 2.6) into CNF through De morgan's distribution law, this would produce 2^n clauses. This is not practical when n grows. If we use one of Tseitin transformations (Equation 2.8) instead, a DNF (Equation 2.6) can be converted into CNF (Equation 2.7) with only $3 \times n + 1$ clauses and new n variables.

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n) \tag{2.6}$$

$$z_1 \vee z_2 \vee \cdots \vee z_n, \text{ for each } i, \; z_i \leftrightarrow (x_i \wedge y_i) \tag{2.7}$$

$$z_i \leftrightarrow (x_i \wedge y_i) \Longleftrightarrow (x_i \vee \neg z_i) \wedge (y_i \vee \neg z_i) \wedge (\neg x_i \vee \neg y_i \vee z_i) \tag{2.8}$$

In the rest of this section, we address search algorithm and SAT Competitions through subsections repectively. Firstly, we describe the DPLL, the initial search algorithm for SAT solvers. The design of DPLL is very simple, and the understanding of DPLL would help to grasp the algorithms in modern SAT solvers. Modern SAT solvers improved their algorithms based on DPLL with several technical elements such as propagation, clause learning, and branching heuristics. Details of elements in modern SAT solvers would be described in the following sections. Secondly, we mention about SAT Competitions which is held biennially as an event in the SAT conferences. The existence of SAT Competitions make SAT researchers collaborate and compete with each other, and researchers are able to share their ideas and challenge newly updated benmark areas.

2.2.1 Search algorithm

Search algorithms are divided into two categories of incomplete and complete search.

Many of incomplete SAT solvers perform SLS algorithms. Initial approaches were attempts to improve the random walk procedure by flipping assignments of variables. GSAT [24] is a representative incomplete SAT solver and recursively generate random assignments and flip the most influential assignment of variable which gives the largest increase in the number of satisfied clauses. To improve efficiency of random walk, tabu search is applied in SAT solvers [25]. These solvers cannot prove unsatisfiability for given formulas, but they can find a solution for satisfiable formulas efficiently from several application areas [26][27].

Complete SAT solvers correspond to both satisfiable and unsatisfiable formulas. Initially introduced search algorithm was DPLL algorithm. This algorithm performs backtracking search by traversing tree through depth-first search (DFS). Details of DPLL procedure is displayed in Algorithm 1. For a SAT formula Γ, a SAT solver determines whether this formula is satisfiable or unsatisfiable through a DPLL algorithm returning a value of *TRUE* or *FALSE* respectively. The DPLL algorithm performs propagation steps while there exist unit clauses (line 2-8), because unit clauses force assignments of variables in them. Figure 2.1 shows an example of propagation. Propagation reduces a formula by eliminating satisfied parts. We marked satisfied parts with a gray color. By assigning *TRUE* to a variable x_0, clauses including a literal x_0 are satisfied and clauses including a literal $\neg x_0$ are shrinked. After this propagation, x_1 is picked and *TRUE* value

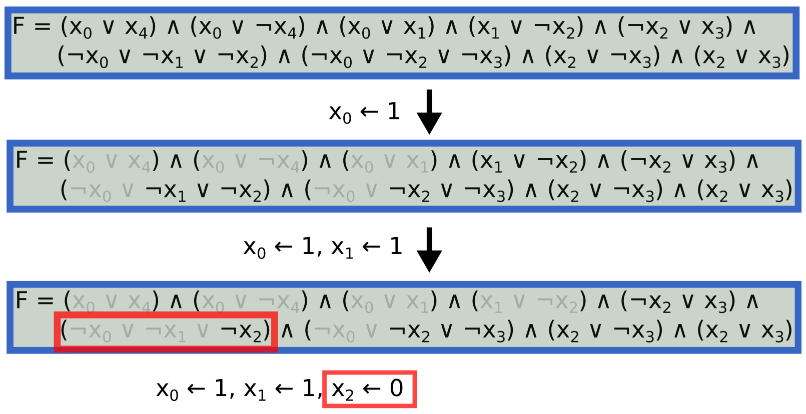

F = (x0 ∨ x4) ∧ (x0 ∨ ¬x4) ∧ (x0 ∨ x1) ∧ (x1 ∨ ¬x2) ∧ (¬x2 ∨ x3) ∧
(¬x0 ∨ ¬x1 ∨ ¬x2) ∧ (¬x0 ∨ ¬x2 ∨ ¬x3) ∧ (x2 ∨ ¬x3) ∧ (x2 ∨ x3)
x0 ← 1
x0 ← 1, x1 ← 1
x0 ← 1, x1 ← 1, x2 ← 0

Fig. 2.1: Example of unit propagations

is assigned and propagated, and this selection induced a new unit clause $(\neg x_2)$ and a new propagation procedure is performed. When a solver picked x_0 and x_1 there were no unit clause in a formula F. In this time, variables are selected arbitrarily, these decisions can be performed through a branching heristic. When propagation steps are done in the DPLL algorithm, a solver checks whether Γ is empty (line 9-10). Empty Γ means current assignment satisfied initial SAT formula Γ. If Γ is not empty, a solver pick a literal for tree search through a branching heuristic (line 12-13). Selecting a proper literal significantly affects on the search efficiency. Figure 2.2 shows an example of two different search trees for the same formula F. Red boxes correspond to the conflicts. For example, we assign x_0 to *FALSE*, then x_4 is forced to both *TRUE* and *FALSE*. This is a conflict. Therefore a search backtracks to x_0 and assigns *TRUE* for x_0. When we compare search trees 1 and 2, tree 1 seems to look more efficient than tree 2. Actually tree 2 contains two forms of tree 1. Choosing x_4 makes search worse in this example. We may explain the different of choices between x_0 and x_4 on a graph. Figure 2.2 describes a graph from a formula F. Each node represents each variable, and we hypothesized its community structure. As a purple community is constituted by only a variable, x_4, we have to find a solution from x_0. On the other hand, x_0 is connected to both communities. Therefore, we can read from a graph that x_0 might be the best selection to find a solution in F. However, selection of variables through the analysis of a graph is not a good idea during searches. A graph changes dynamically through the assingnments or addtion of clauses. A formula F can be updated to F' through clause learning by a resolution, details will be discussed following sections. A graph is also changed through the addition of a clause $(x_1 \vee x_3)$.

Algorithm 1 DPLL algorithm.

```
    Input: SAT formola $\Gamma$, literals $L$
    Output: TRUE or FALSE
 1: function DPLL($\Gamma$, $L$)
 2:     loop
 3:         if A unit clause $u$ exists in $\Gamma$ then
 4:             ($\Gamma$, $L$) $\leftarrow$ unitPropagation($\Gamma$, $L$, $u$)
 5:         else
 6:             break
 7:         end if
 8:     end loop
 9:     if $\Gamma$ is empty then
10:         return TRUE
11:     end if
12:     Choose a literal $l$ from $L$
13:     $L \leftarrow L \setminus \{l\}$
14:     return DPLL($\Gamma \cup \{l\}$, $L$) or DPLL($\Gamma \cup \{\neg l\}$, $L$)
15: end function
```

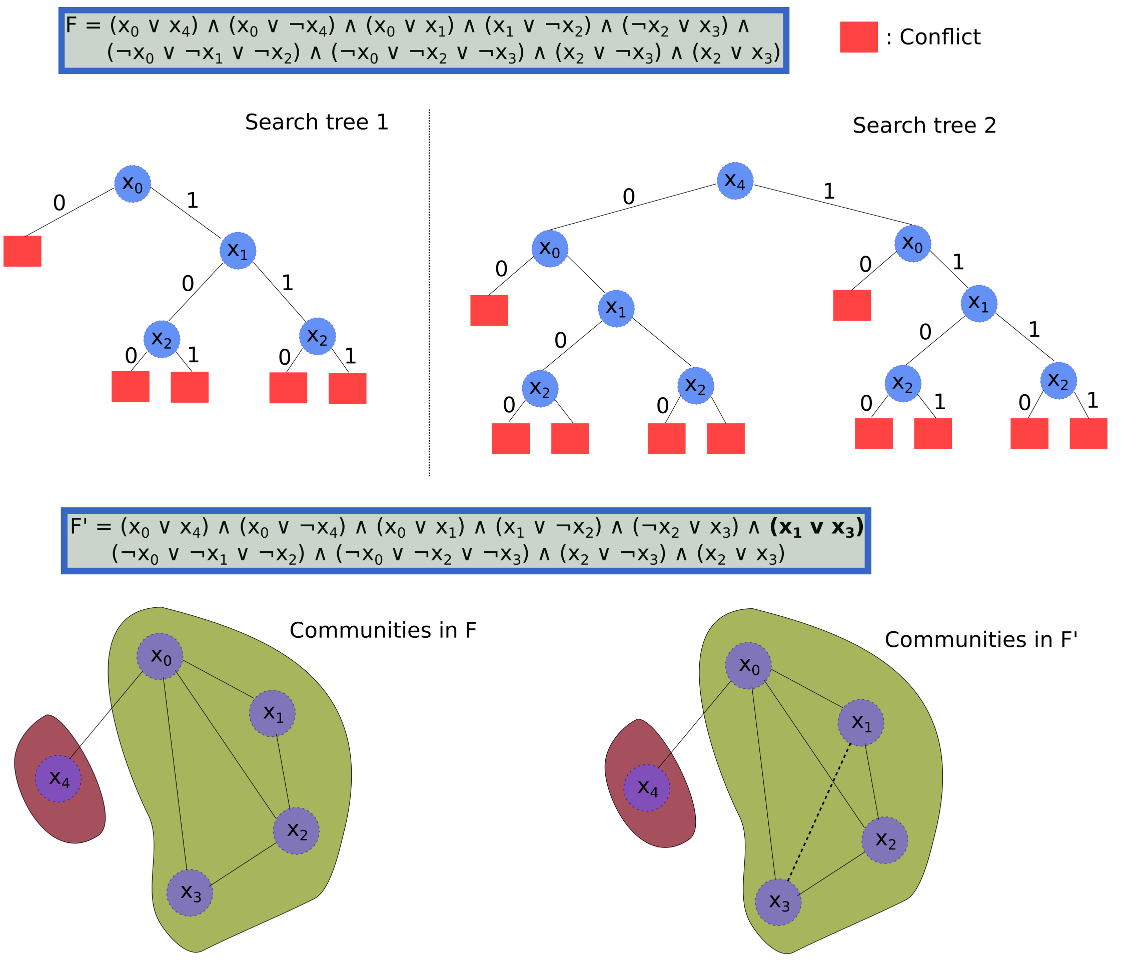

$F = (x_0 \lor x_4) \land (x_0 \lor \neg x_4) \land (x_0 \lor x_1) \land (x_1 \lor \neg x_2) \land (\neg x_2 \lor x_3) \land (\neg x_0 \lor \neg x_1 \lor \neg x_2) \land (\neg x_0 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor \neg x_3) \land (x_2 \lor x_3)$
🟥 : Conflict
Search tree 1
Search tree 2
x_0
x_1
x_2
x_4
0
1
$F' = (x_0 \lor x_4) \land (x_0 \lor \neg x_4) \land (x_0 \lor x_1) \land (x_1 \lor \neg x_2) \land (\neg x_2 \lor x_3) \land \mathbf{(x_1 \lor x_3)}$ $(\neg x_0 \lor \neg x_1 \lor \neg x_2) \land (\neg x_0 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor \neg x_3) \land (x_2 \lor x_3)$
Communities in F
Communities in F'
x_0
x_1
x_2
x_3
x_4

Fig. 2.2: Diverse search trees and community structures through the changes of a formula

To improve the initial DPLL approach, several techniques are proposed and improved. Clause learning based on conflicts, efficient data structure for propagation and the agile branching heuristic are representative ones, and details of them are described in the following sections. There are other technuques called restart and simplification. The restart is discussed shortly in Section 2.7. We introduce simplification methods here.

We first introduce a resolution [28] technique. Equation 2.9 shows an example of a resolution. C, D, C_1, and D_1 indicate a clause respectively. When a clause C includes a literal x and D includes $\neg x$, a resolution of C and D is obtained by $C_1 \vee D_1$.

$$\begin{gathered} C = C_1 \vee x,\, D = D_1 \vee \neg x \\ C \otimes_x D = C_1 \vee D_1 \end{gathered} \tag{2.9}$$

A clause generated by two clauses containing complementary literals is called resolvant. We expressed a reslovant of C and D against x using $\otimes_x$. The notion of resolution is used in other simplifications, and the chain of resolutions is used to learn new clauses in the CDCL SAT solvers.

Representative simplification methods are bounded variable elimination (BVE), subsumption elimination (SE), self-subsuming resolution (SSR) [29], and blocked clause elimination (BCE) [30]. We give an equation of BVE at Equation 2.10 as an example. F_x is a set of clauses including a literal x and $F_{\neg x}$ is a set of clauses including a literal $\neg x$. A variable x can be eliminated from a SAT formula, if a solver replace $(F_x \cup F_{\neg x})$ to a resolvant set of $(F_x \otimes_x F_{\neg x})$. This is variable elimination (VE) and we skip the proof of this, but VE preserve the satisfiability. An equation 2.10 shows the condition to perform BVE. The condition indicates that BVE is applied only when a total number of clauses in a SAT formula can be reduced through this simplification.

$$\text{If, } |F_x \otimes_x F_{\neg x}| \leq |F_x \cup F_{\neg x}|,\; F \rightarrow (F \backslash (F_x \cup F_{\neg x})) \cup (F_x \otimes_x F_{\neg x}) \tag{2.10}$$

Generally simplification methods are performed as a preprocessing in the SAT solvers, because they require a lot of calculation time and applying them does not ensure SAT solvers to find a solution rapidly. Lingeling [31] applied these methods adequately not only in preprocessing but during search, called *inprocesssing*, and is showing nice performances since SAT Competition 2014.

Our research in this paper is designed for the complete SAT solvers. However, we are considering the stochastic local search approach can be useful for complete solvers. In Chapter 6, we propose a history map, apply this to a complete SAT solver, and propose a random walk method.

2.2.2 SAT Competitions

SAT Competitions are organized as a satellite event of International Conferences on Theory and Applications of Satisfiability Testing (SAT conferences). Competitions are held annually from 2002 to 2005, biennially from 2007 to 2013, 2014 and 2016 [11]. Main objectives of competitions are gathering new challenging benchmarks, sharing ideas, and evaluating new approaches; thus SAT solvers are getting more efficient through the competitions.

Details of competitions such as tracks or evaluation methods are consistently changing over time. Categories of instances did not change, there are 3 types of instances those are categorized by application, crafted and random throughout the competitions. Crafted instances are intentionally designed hard to solve for SAT solvers. The name of this category changes sometimes, and it is called such as hard-combinatorial or handmade. We introduce detailed tracks in SAT Competition 2016.

- Agile Track: A newly introduced track from SAT Competition 2016. Solve each benchmark within a short time T. T is set to 60 seconds.
- NoLimit Track: A newly introduced track from SAT Competition 2016. No obligation for submitting the source code or emitting an unsatisfiability proof. Portfolios of existing solvers are allowed.
- Random Track: Solve randomly generated instances.
- Main Track: Solve 300 application instances and 200 crafted instances. Time limit is 5,000 s. Emitting an unsatisfiable proof is required.
- Parallel Track: Using the same 500 benchmarks used in Main Track. Fourty-eight CPU (hyper-threading) environment is provided with 64GB RAM. Time limit for each instance is 5,000 s.
- Incremental Track: Solve application instances generated from four different cate-

gories. Assess by averaging the ranks from each category.

- Glucose Hack Track: Modification of Glucose 3.0 less than 1000 characters.

There were limitations of number of submissions. Each participant can submit up to four different sequential solvers, two different parallel solvers and one Glucose Hack solver. We submitted four different sequential solvers based on Glucose 3.0 and two versions of parallel solvers. We obtained silver medal and bronze medal in Agile Track and got the best Crafted benchmark solver in Main Track. A Glucose 3.0 with *tie-breaking* showed its efficiency in Agile Track, and a Glucose 3.0 with a hybrid branching heuristic using two branching heuristics VSIDS [14], CHB [15] and *tie-breaking* method showed it efficiency for the crafted benchmarks. Detailed results and downloading links of source codes are available on SAT Competition 2016 web page [32].

2.3 Propagation

We already explained the notion of the propagation in Section 2.2.1. In this section, we introduce the smart data structure for propagations which has largely contributed to the performance of SAT solvers. The propagation, also called Boolean constraint propagation (BCP), process accounts for 70-90% of the CPU time in the modern SAT solvers [33]. Therefore, even a minor improvement of propagation can speedup a SAT solver largely when we consider Amdahl's law.

Initial approaches for propagation was performed directly updating adjacent lists of literals in clauses and lists of clauses related to a literal. Hiding satisfied clauses and literals in clauses as we shown in Figure 2.1 was the initial approach. There were also counter-based approaches [34][35] by counting satisfied and unsatisfied literals for each clause. However, these approaches need exact updates of hiding, recovering and counting in both assignments and backtracks. And the exact updates need to traverse all clauses requiring a lot of time.

Traversing all the remained clauses for each propagation is too time-consuming, and most of updates through propagations does not affect on the search directions because they have more than two variables unassigned after the propagations. If we only concentrate on the clauses those have a possibility to turn into a unit clause, a lot of time for traversing

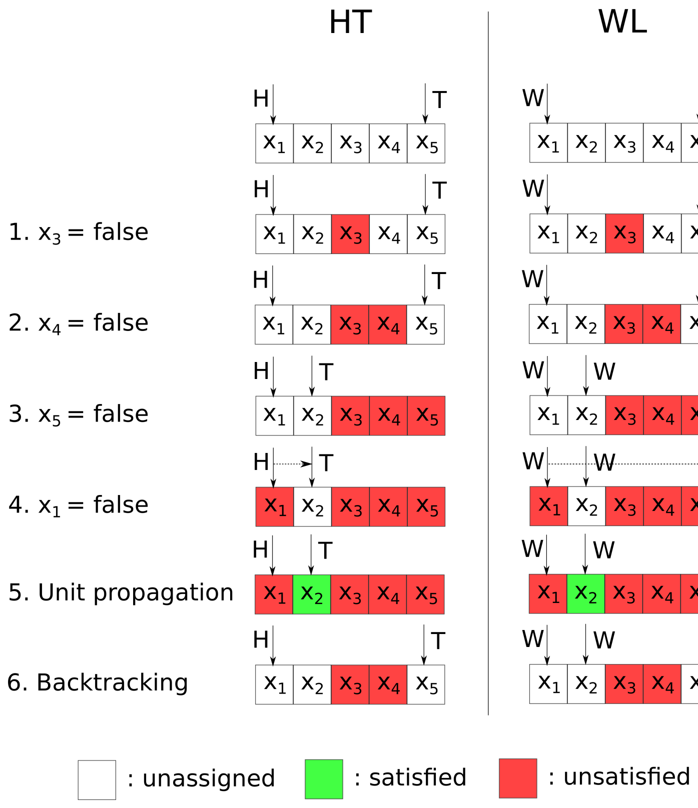


Fig. 2.3: Example of Lazy data structures

clauses would be saved. Therefore, a lazy data structure was proposed in the SATO [36] and Chaff [14]. These data structures monitor only two literals in each clause as references. SATO uses the Head/Tail (HT) structure and Chaff proposes Watched Literals (WL). The large difference between them is the existence of an order relation between the two references. Figure 2.3 illustrates an example of HT and WL. There are six steps from #1 to #6 in Figure 2.3. There is no update of references at #1 and #2, because they prefer lazy updates. Updates are performed when only either of references is assigned. At #3, a reference is assigned *FALSE* and it's position is updated by finding and moving to an unassigned literal. At #4, there is a difference between HT and WL. H is positioned at the left side of T in HT. The order relation between two W does not exist in WL. Therefore, H stop traversing a clause at X_2 and W at X_1 traverse all literals in a clause. Unit propagation is performed in HT and WL respectively at #5, because they recognized current clause is unit clause in #4. After a conflict, backtracks are performed at #6. In HT, references have to be recovered at previous position to maintain the order relation. However, WL does not need an update when backtracks performed.

Modern SAT solvers use WL or its variants and they are often called watched literals or two-literal watching. Binary and ternary clauses have a high probability to become unit clauses, thus these clauses have a priority in propagation procedures to guide early propagations [37]. There also exist researches with the consideration of data structures to enhance a cache performance [33][38].

Modern SAT solvers use lazy data structures for propagation, but they do not update enough information and this might limit the inprocessing performance based on rules of inferring binary/unit clauses [39].

In this paper, we utilized watched literals in our initial approach of new branching heuristic discussed in Chapter 4.

2.4 Clause learning

We discussed DPLL, the basic backtracking search algorithm and several simplification methods in Section 2.2.1. Modern SAT solvers learn new clauses during search through DPLL and some simplifications. They are called conflict-driven clause learning (CDCL) [13] solvers. The main objective of clause learning is to avoid "duplicates" already explored during the search by adding them to an original SAT formula working as new constraints to prune redundant areas.

CDCL learns a new clause through the analysis of an implication graph when a conflict occurs. Generally, an implication graph is well-used to solve a 2-SAT problem with a linear time algorithm [40]. Each node in a graph corresponds to an assignment of a variable with its decision level. A new node can be generated by decision or induced from the previous decisions and propagations.

Figure 2.4 shows an example of clause learning. Consider a SAT formula consisted of six clauses of C_1 to C_6. Each node in the implication graph has a variable and its (value)@(decision level). If there is no unit clause in a SAT formula, a new variable is picked and this is a decision process. The previous decisions are displayed with grey color. In each decision level, a variable is selected and assigned, and propagations are progressed. The current decision level in Figure 2.4 is a 4, and x_3 is selected and assigned by 0 (*FALSE*). Decision node at the current decision level is green-colored. After the assignment of x_3, the chain of propagations are performed. For example, x_4 is forced to

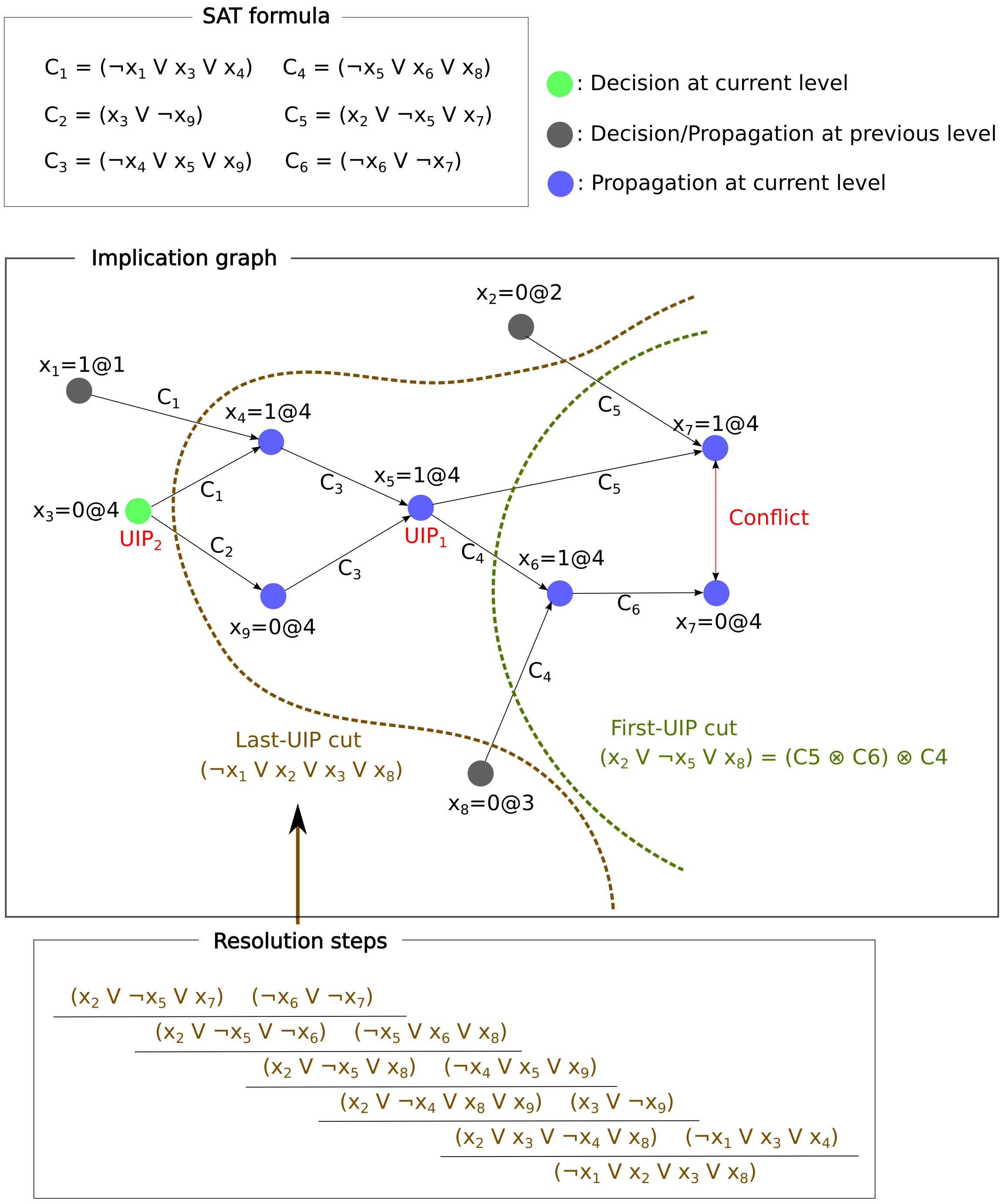

SAT formula
$C_1 = (\neg x_1 \lor x_3 \lor x_4)$
$C_2 = (x_3 \lor \neg x_9)$
$C_3 = (\neg x_4 \lor x_5 \lor x_9)$
$C_4 = (\neg x_5 \lor x_6 \lor x_8)$
$C_5 = (x_2 \lor \neg x_5 \lor x_7)$
$C_6 = (\neg x_6 \lor \neg x_7)$
● : Decision at current level
● : Decision/Propagation at previous level
● : Propagation at current level
Implication graph
x_2=0@2
x_1=1@1
C_1
x_4=1@4
C_5
x_7=1@4
C_3
x_5=1@4
C_5
x_3=0@4
C_1
UIP$_2$
C_2
UIP$_1$
Conflict
C_4
x_6=1@4
C_3
C_6
x_7=0@4
x_9=0@4
C_4
Last-UIP cut
$(\neg x_1 \lor x_2 \lor x_3 \lor x_8)$
x_8=0@3
First-UIP cut
$(x_2 \lor \neg x_5 \lor x_8) = (C5 \otimes C6) \otimes C4$
Resolution steps
$(x_2 \lor \neg x_5 \lor x_7)$ $(\neg x_6 \lor \neg x_7)$
$(x_2 \lor \neg x_5 \lor \neg x_6)$ $(\neg x_5 \lor x_6 \lor x_8)$
$(x_2 \lor \neg x_5 \lor x_8)$ $(\neg x_4 \lor x_5 \lor x_9)$
$(x_2 \lor \neg x_4 \lor x_8 \lor x_9)$ $(x_3 \lor \neg x_9)$
$(x_2 \lor x_3 \lor \neg x_4 \lor x_8)$ $(\neg x_1 \lor x_3 \lor x_4)$
$(\neg x_1 \lor x_2 \lor x_3 \lor x_8)$

Fig. 2.4: Example of clause learning

1 because $x_3 = 0$ and $x_1 = 1$ simplifying $C_1 = (x_4)$. During propagations, a conflict is found because $C_5 = (x_7)$ and $C_6 = (\neg x_7)$. A solver can learn a clause from this conflict to prevent repeating the same mistake. A new clause is obtained through a chain of resolutions we introduced in Equation 2.9.

Different resolution steps generate learned clauses. Finding a proper cut for each resolution would be difficult work. Generally, the notion of unique implication point (UIP) is used for cuts. A unique implication point (UIP) indicates a node in an implication graph which is induced at the current decision level and all paths from the decision node to the conflict node pass through it. There are two UIPs of UIP$_1$ and UIP$_2$ in Figure 2.4. The order of UIPs are sorted by distances from conflict nodes by the ascending order. Resolutions are performed from a conflict, and we call a UIP encounters for the first time during a conflict analysis a first-UIP. A UIP$_1$ is the first-UIP and a UIP$_2$ is called the last-UIP. A Figure also cantains the detailed resolution steps at UIPs. A UIP cut divides an implication graph into two parts of the reason side and the conflict side. The right side of a UIP cut is the conflict side and the left side and the UIP itself are included in the reason side. Initially, the last-UIP [41] was used to learn clauses, but the first-UIP [42] showed its efficiency based on experimental results, and now many modern SAT solvers learn clauses through first-UIP cuts.

Once a learned clause is obtained, backtracks are performed, often called backjumping procedure. A backtrack level is determined through the Equation 2.11.

$$\begin{gathered} \textit{backtrack level} = \arg\max_{x \in (C \setminus C')} \textit{level}(x) \\ (C\text{: learned clause, } C'\text{: set of literals induced at current decision level}) \end{gathered} \tag{2.11}$$

Therefore, if a SAT solver uses first-UIP in Figure 2.4, it compares decision levels of x_2 and x_8, and backtracks are performed until it reaches decision level 3.

We explained the learning scheme of CDCL solvers. However, the number of potential combinations of new clauses is $n! \times 2^n$ for a SAT formula with n variables and a new clause is found on each conflict. If a SAT solver accumlates learned clauses persistently without any restrictions, the increased size of learned clause database makes each propagation time longer, a solver gets slower and slower, and finally reaches to the memory overflow problem. Therefore, periodical reductions of a learned clause database are required. When we reduce the size of a database, we want to maintain more valuable clauses and remove low-informative clauses. To achieve this, there exist several size-bounded approaches [?] and relevant-bounded approaches [41][43]. Size-bounded approach is extremely simple. The i-size-bounded approach erases clauses consisted of more than i literals. This approach is quite reasonable because when a clause get shorter, the probability of unit propagation would get higher. However, sometimes long clauses are required for the unsatisfiable proof of an SAT instance.

The i-relevant approach removes learned clauses when more than i literals in them are satisfied or unassigned by current assignment. The idea is, if many literals in them are already satisfied or unassigned, the probability of them to become a reason for other conflicts in the near future would be very low. We give an example to support the idea of relevant-bounded approach.

$$\text{partial assignment P} = \{\ x_1 = TRUE, x_2 = TRUE, x_3 = TRUE\ \}$$
$$C_1 = (x_1 \vee x_2 \vee x_3)$$
$$C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee x_5)$$

If we consider clauses C_1 and C_2 under assignment P, then C_1 is always satisfied and C_2 is a binary clause. Therefore, C_2 would be more useful under P.

Recently, clauses are assessed using literal blocks distance (*LBD*) [44]. We introduce *LBD* because *LBD* index is widely used in modern SAT solvers to assess clauses. Many solvers use *LBD*s of learned clauses instead of their sizes or use a hybrid policy using both *LBD*s and sizes.

Figure 2.5 shows several examples of *LBD*. A clause C_1 is an example of a learned clause we showed in Figure 2.4. *LBD* assesses learned clauses with the number of blocks. Each variable have its decision level, and variables having the same decision level are integrated in the same block. Variables in the same block have the same decision level and this indicates they were induced from the same unit propagation. Therefore, *LBD* idea considers that they might be propagated at the same decision level again in the future. Especially, a clause with $LBD = 2$ is called *glue clause* and considered very important. If we evaluate clauses using *LBD*s, clauses C_1 and C_2 are assessed as the same. Modern solvers assess learned clauses *LBD* and also their sizes. *LBD* measure is also used for clause sharing policies for parallel SAT solvers. Details are discussed in Section 2.6.

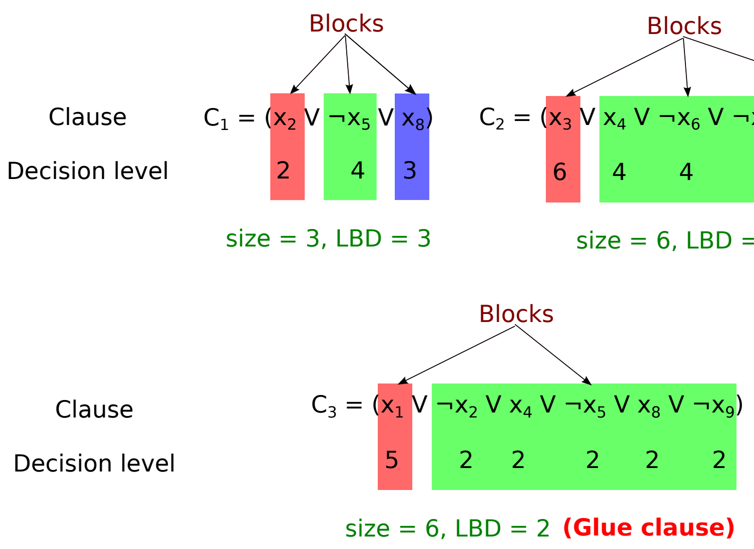


Fig. 2.5: Example of *LBD*

In this paper, we propose a new branching heuristic with consideration of interplay between the clause learning scheme and the branching heuristic in a SAT solver. We are considering a SAT solver works efficiently because of well-balanced combination of heuristics. Therefore, when we improve a heuristic or design a solver, the interplay of subparts should be considered deeply. An *LBD* index is utilized both in our sequential and parallel solvers.

2.5 Branching heuristic

A SAT solver performs a tree search and pick a variable for the next branch from the unassigned variables. Which path a solver chooses significantly affects the search efficiency. Initial branching heuristics were concentrated on reducing the size of the search space, when DPLL search algorithm was mainly used for SAT solvers.

Bohm's heuristic [45] selects a literal with the maximal vector $(H_1(x), H_2(x), ..., H_n(x))$ under the lexicographic order. Each $H_i(x)$ is computed by Equation 2.12, where $h_i(x)$ is the number of unresolved clauses with i literals, which contain a literal x.

$$H_i(x) = \alpha \times max(h_i(x), h_i(\neg x)) + \beta \times min(h_i(x), h_i(\neg x)) \tag{2.12}$$

Dynamic literal individual score (DLIS) [46] finds variables x and y, each of them maximizes $C_{x,p}$ and $C_{x,n}$ respectively, where $C_{x,p}$ and $C_{x,n}$ are the number of unresolved clauses in which x appears positively and negatively respectively. If $C_{x,p} > C_{y,n}$ assign x to *TRUE*. Otherwise, *FALSE* value is assigned to y. There exist variants of DLIS [47][48].

Maximum occurences in clauses of minimum size (MOM) [49] gives preferences to the unresolved smallest clauses. Each variable x is computed by Equation 2.13, where $w(x)$ is the number of unresolved clauses, which contain a variable x. several variants of MOM were also proposed [50][51].

$$H(x) = (w(x) + w(\neg x)) \times 2^k + w(x) \times w(\neg x) \tag{2.13}$$

Heuristics mentioned above are quite expensive because they have to traverse all the unresolved clauses. Furthermore, the CDCL [13] technique with the lazy clause watching techniques [36][14] makes these heuristics more expensive, because a solver with CDCL learns new clauses, increased number of clauses makes a propagation time longer, and the lazy clause watching techniques does not reflect all propagations. However, these techniques are essential, because the propagation is the most expensive operation in SAT solvers.

For the agile score updating for branching heuristics in CDCL solvers, the variable state independent decaying sum (VSIDS) [14] is proposed. In the original VSIDS, each variable has a counter, and variables in a learned clause are incremented by 1 when a conflict occurs and conflict analysis is performed. At each decision, a variable with the highest counter is selected. However VSIDS in modern SAT solvers increment all variables those are related to obtain a learned clause through resolution steps [28]. Incremented counters are decayed at regular intervals for intensive search by emphasizing recently learned clauses. In original VSIDS, counters are divided by 2 on every 256 conflicts. However modern SAT solvers decay scores more frequently. Many solvers decay scores on every conflicts. Thus variables related to the most recent conflict are bumped [52], and the others are decayed.

Normalized VSIDS (NVSIDS) [53] uses a decay factor f $(0 < f < 1)$. When a conflict occurs, decay all variable first. Nextly, add $1 - f$ to variables related to resolution steps.

Exponential VSIDS (EVSIDS) [53] bumps variables related to resolution steps to $s + g^i$ with a bump factor g $(1.01 < g < 1.2)$, where i is a conflict index. A design for avoiding overflow problems is required to achieve EVSIDS, because g grow fast. For example in

MiniSat, when a score over 1^{100} is found, a rescoring is performed. As $g = 1.05$ as default, the rescoring interval is around 4720 conflicts. The order of variables is maintained as the same in both NVSIDS and EVSIDS, but, EVSIDS works faster than NVSIDS because it does not need to update scores of all variables. EVSIDS is one of VSIDS variants, but now this one is called VSIDS and utilized into many state-of-the-art solvers. In this paper we call EVSIDS to VSIDS.

ACIDS [54] proposes a different bump factor, and author insists this factor reduces overflow frequency, but decaying process will be slowed down.

Several researchs [55][56] gave the attempts to beat VSIDS, strictly speaking EVSIDS. However, VSIDS showed its robustness and still widely used in modern SAT solvers.

Recently proposed braching heuristics [15][57] relatively work well compared to VSIDS. Conflict history-based branching heuristic (CHB) [15] uses the concept of the reinforcement learning and update scores based on rewards calculated by the conflict history. Authors of CHB also proposed learning rate branching (LRB) [57], proposing a learning rate between an assignment and an unassignment for an attempt to maximize the quantity of learned clauses. These branching heuristics might replace VSIDS, if they show their efficiencies continuously through new benchmarks.

When a branching heuristic chooses a variable, not a literal, we have choose a literal from a variable itself and its negation for the assignment. A smart policy of $TRUE/FALSE$ assignment for each variable may improves the efficiency of a search. In MiniSat [58], the negative assignment is chosen as the default. In ManySAT [59], each literal has the number of occurrences in learned clauses, and after a selection of variable x in a decision heuristic, a solver set x to $TRUE$, if the number of occurrences of x is larger than that of $\neg x$.

Phase saving [60] map saves the last $TRUE/FALSE$ assignment for each variable. When a conflict occurs at a decision level A, a SAT solver backjump to a decision level B to avoid the same conflict. Then we cancel assignments of variables between A to B. However, assignments before the cancelations might include useful information or partial solutions. By saving last assignments, they can be reused in a branching heuristic.

This paper have several connections with branching heristics. In Chapter 3, we perform several experiments related to VSIDS and obtain several knowledges about the relationship between the diversification of branching heuristics and the performance variation of the SAT solvers. In Chapter 4, we propose a new branching heuristic that can be utilized to to intesify the search by enhancing the interplays between a branching heuristic and a clause learning scheme. In Chapter 5, we combine several branching heuristics to diversify a SAT solver. A phase saving technique is utilized in Chapter 6 for the construction of history map in a parallel SAT solver.

2.6 Parallel SAT solver

For the last two decades, there were tremendous improvements in SAT solvers with several techniques we discussed sections above. However, it has become hard to achieve a remarkable improvement in sequential solvers. And at sort of the propitious moment multicore hardware and cloud computing have evolved drastically. Naturally researchers gave attempts to design parallel SAT solvers for achieving notible improvements. There are mainly two approaches to design a parallel SAT sover, which are the divide and conquer and the portfolio. Divide and conquer approaches partition the search space into subspaces and each worker searches each subspace which is completed seperated from other subspaces. Portfolio approaches do not divide the search space and all workers find a solution in the entire search space with different policies. Divide and conquer approaches are explained briefly, because our parallel approaches are aim for SAT solvers with portfolio approaches.

Initial approaches were performed with the divide and conquer paradigm, and solvers with this paradigm mainly utilized the guiding path concept [61]. Search space is divided into subspaces and have to be managed among workers. Therefore, divide and conquer solvers generally adopt the master-slave concept. If an idle worker (subspace search has finished) appears, another subspace is allocated to it [62][63]. However, finding an efficient load balancing strategy was difficult in their initial divide and conquer approaches.

A portfolio approach was the next to appear. Applying Economics for the hard combinatorial problems to reduce a risk were tried in the late 1990s [64][65]. Sharing of learned clauses is very important in the portfolio-based parallel solvers, because they do not divide the search space, and it is an undesirable situation that a worker searches in the areas witch are already checked by other workers. As an effort to reduce the redundancies and diversify search, a portfolio is constituted by assigning different search policies for each worker.

ManySAT [66] designed a portfolio for a parallel SAT solver with four workers and obtained first rank in the parallel track of the 2008 SAT-Race. Since then, a portfolio approach have become a mainstream for designing the parallel SAT solvers. ManySAT 1.0 allocated different strategies, the restart policy, branching heuristic including *TRUE*/*FALSE* assignment policy, learning scheme and clause sharing policy, for each worker.

ManySAT 1.1 [67] dynamically adjusts the limitation policy of clause sharing using their size through AIMD (Additive-Increase-Multiplicative-Decrease) feedback control-based algorithm [68]. ManySAT 1.0 shared all clauses with a size of under eight among workers. ManySAT 1.1 measures a quality of a learned clause between pairs of workers when it is shared, and decrease/increase size limitation when the quality is bad/good, respectively.

Glucose-syrup [69] does not share the learned clauses immediately. It prepares a space called "probation" and keeps shared clauses in it and watches them. When a shared clause in a probation is falsified during search, this is considered as useful and added to the learned clauses database. This lazy clause sharing gives an attempt to reduce the useless sharings of learned clauses which might deteriorate the propagation rate of a SAT solver.

Penelope (parallel *LBD* psm solver) [70] uses a psm (progress saving based quality measure) [71] indicator to adjust freezing and activating of learned clauses. The psm measures intersection of literals in a clause and the current value assignment in a solver. A clause with a low psm indicates that many literals in it satisfies itself. In this case, freeze a clause because it might not be utilized for a unit propagation or a conflict in the near future. On the other hand, if a psm of a clause is high, activate this. They determined policies for exporting and importing clauses using a freezing concept.

There are several criticisms about the scalability problem about portfolio approaches [72][73]. Many of divide and conquer approaches [74][75][76] are showing their scalablities. On the other hand, there also exists a research showing the possibility of scalability for portfolio SAT solvers [77]. However, what we can insist is that the current portfolio solvers cannot scale. Portfolio solvers work efficiently, becuase they share learned clauses actively based on their different search policies. This cannot be performed for massively parallel environments. In this paper, we propose a data structure that is applicable for combinatorial search to adjust the diverficiation and intensification. We propose a history map which requires only a small memory; thus can be applied for parallel SAT solvers in the massively parallel environments.

2.7 Diversification and intensification

Branching heuristic, we mentioned in Section 2.5, is the most representative technique for intensive search. A branching heuristic attempts to find a literal that seems to influence a lot, such as highly reducing the size of a search space or inducing huge amounts of propagation chains. Many of branching heuristics are also designed to prioritize the recently utilized variables for further intensive search. Applying phase saving technique [60] also strengthen the search intensification.

An intensive search sometimes make a SAT solver wanders into one of deserts containing no solutions [78]. As a result, some instances have remarkable variabilities in the time required to find a solution [79]. To escape from a desert, avoid *heavy-tailed* behavior [79], and diversify search, several stochastic approachs have been proposed.

Restart [80] is one of the most representative techniques for search diversification. When a SAT solver reaches a certain number of conflicts, it restarts the search by canceling all the assignments and backtracking to the root of the search tree. A solver might escape from a desert through restarts. As we abovementioned, they exist time variabilities for SAT instance to be solved. Applying restart technique may reduce the extent of variabilities.

Several approaches have been proposed to achieve the efficient restart intervals. Luby restart [81], nested restart [82], and others (geometric, arithmetic) [52][83] applied static methods based on the number of conflicts in a SAT solver. Solvers with these restarts only moniter the conflict numbers and do not consider other features such as recent conflict rates, current decision level, and so on. An appropriate cutoff value for restart is not exist. However, this is a difficult work, becuase we cannot measure the possibility of the existence of a solution at the current search status. Generally, it is believed that hard instances need slow restart and easy instances need rapid restart. However, when a solver searches in a desert of a hard instance, rapid restart would be required. Constructing an efficient and general dynamic restart policies looks very difficult. Several researches [53][84][85][86] gave the attempts to propose dynamic restarts. Measuring agility [53] shows quite interesting ideas. They measured the agility of a SAT solver based on the rate of recently flipped assignments by using phase saving [60] map. The idea of Glucose restart [87] looks also good. They monitored the average *LBD*s [44] through recently learned clauses and utilized this value for restart strategy. Their ideas is that the low average of *LBD*s indicates the production of good clauses. Therefore, when *LBD*s get higher, a solver restarts.

Random initial scoring for branching heuristics and random branching with fixed probability [58] had been adpoted as the default features to achieve search diversifications in many solvers, but they are deprecated in modern SAT solvers because they seem to deteriorate a balance between the diversification and intensification resulting the performance of SAT solvers worsen.

Adjusting between the diversification and intensification gets more difficult when we discuss parallel SAT solvers. In parallel solvers, we should consider not only the balance in a worker and also the balance among workers. We skip the divide and conquer solver and concentrate on portfolio-based SAT solvers. They produce diversity by assigning different policies for each worker such as a restart policy, clause learning scheme, a decaying ratio of scores or random selection probability in branching heuristic, and so on. They perform intensive search through clause sharing or designing a topology among workers.

ManySAT 1.5 [88] gave an attempt to find a good master-slave topology for diversification and intensification. Figure 2.6 shows partial 3 topologies from 7 topologies ManySAT 1.5 attempted. Figure 2.6 (b) showed the best results based on their experiments. Clause sharing affects on both diversification and intensification of the search, because by sharing clauses they could avoid nogood areas and also could find new clauses by the aid of shared clauses. Authors of ManySAT 1.5 divided workers into masters and slaves and let a slave searches following the instructions from its master.

In the paper by Lagniez [89], a deterministic assignment selection was used, and each pair of workers was compared by their current assignments through a hamming distance at some control point. If a pair of workers seems to close, pick one of them and invert its assignments.

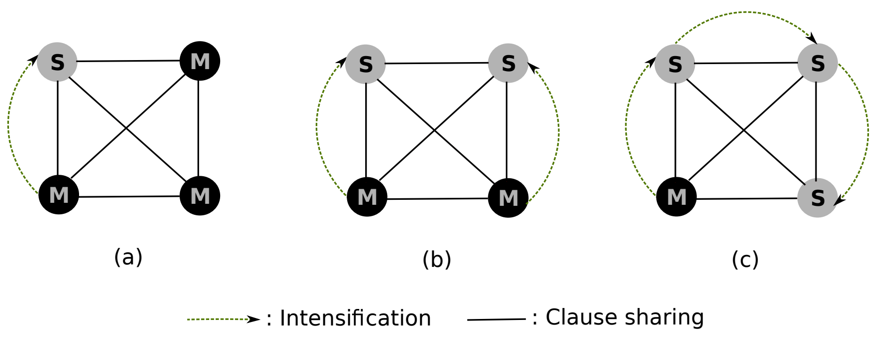


Fig. 2.6: Partial topologies for Diversification/Intensification in ManySAT 1.5

However, researches abovementioned only used the small number of workers, and they proposed very expensive pairwise policies. Their methods cannot be applied in massively parallel environments, because their calculations are not cheap.

Our proposals in this paper are deeply connected to the diversification and intensification. In Chapter 3, we test index shuffling of variables to check whether a modern SAT solver have enough diversity. In Chapter 4, we propose a new branching heuristic to break ties. Our idea is an attempt to select a more valuable variable from ties or an consideration of interplay between a branching heuristic and a clause learning scheme to intensify the search. In Chapter 5, we propose a hybrid branching heuristic for integrating the performance of SAT solvers. This is a first step of our algorithm selection framework. The hybrid model does not diversify search dynamically during search. However, a solver with a hybrid model have a diversify in a sence because it can choose an algorithm dynamically based on an instance basis. In Chapter 6, we propose a approximate history map. This map approximates distribution of visited areas in a search space. We may utilize this map to several scenarios to adjust between the diversification and intensification of search.

Chapter 3

Diversification of search

This chapter discribes our preliminary experiments as an effort to diversifying yet maintaining intensity of search. Methods on this chapter are extremely simple yet gave us knowledges and ideas to progress our researches. Firstly, We implement diversification method in MiniSat 2.2 [58] and observe the variation of execution times. Secondly, we hypothesize why this method produces diversity and verify our assumptions through the experiments. Finally, we implement a parallel solver to check the verification of our diversification method on the parallel solvers.

3.1 Shuffling variables

It is a well known feature of CDCL SAT solvers, that the running times among solvers can vary substantially due to small changes to the input files [90]. And in the early days of the SAT Competitions, the organizers scrambled the benchmarks as a method to compare the performance of SAT solvers [91][92]. However the application of shuffling to produce search diversification is not general, because it is believed that modification of an original formula seems to yield negative effects. We are considering this method might be better than other random methods to diversify search, because shuffling does not deform the hidden structure in its original formula. Experiments based on shuffling gave us several insights and research directions by understanding modern SAT solvers. Especially we shuffled indexs of variables and denote this with *ISoV* (Index Shuffling of Variables). *ISoV* simply shuffles indexes of variables as illustrated in Figure 3.1.

Shuffling is performed through the Fisher-Yates shuffle [93]. After reordering of indexs,

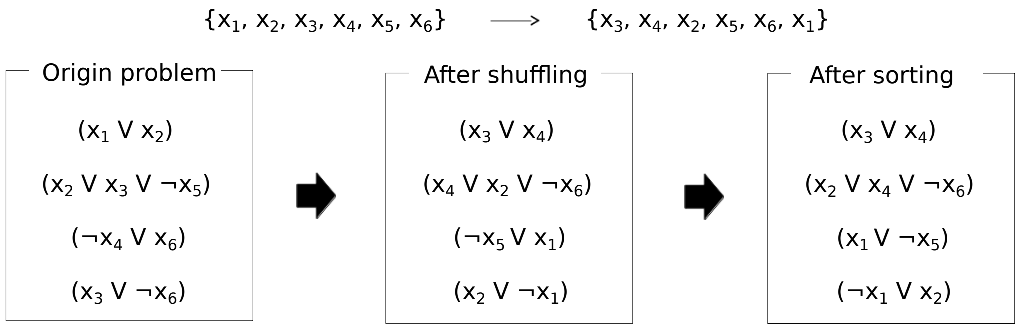


Fig. 3.1: Example of *ISoV*

each clause is sorted by the index order of variables, and the shuffled clause is obtained. The merit of this method is that the *ISoV* does not destroy the structure of the problem, but the changed order of the variables would induce the different searches and the diverse results. We are considering the *ISoV* might be useful to produce diversities in the parallel SAT solvers. Maintaining different searches among workers gets difficult when the number of workers increase. *ISoV* might help to obtain additional diversity among workers. If we set the search policies of all workers identically and only apply different *ISoV* for each worker, the implementation will become extremely simple. *ISoV* is a simple pre-processing method, and can be easily applied to any of parallel SAT solvers.

Why does *ISoV* change execution time?

The changes of orders in the clauses would produce several butterflies, and several of them might effect on the selections in a branching heuristic. When a solver chooses a variable through the evaluation function in a branching heuristic, several of them would have the same highest scores. If a solver breaks these ties randomly, the different variable selections would lead different propagations, different conflicts would be encountered, and different clauses would be obtain during search. Let's consider scores of variables for a branching heuristic are managed by the heap array. our assumption is that applying *ISoV* would change the order of ties in heap array. Therefore, we considered bufferflies from *ISoV* might very similar to the effect of random selection from ties in a branching heuristic. Our another assumption here is *ISoV* does not lower the performance of SAT solvers, because applying this does not deform the traits of the instances. To demonstrate thess assumptions, we did experiments and analyzed them through several SAT instances. The results are described in the next sectiion.

3.2 Experimental results

We describe several experimental results in this section. Firstly, we show search diversification through *ISoV*. Secondly, we analyze why *ISoV* produces the diversity in a SAT solver with assumptions by comparing this with the random *tie-breaking* and other random methods. Finally, we apply this into a parallel SAT solver and verify its utility based on experimental results.

We mainly employed two different execution environments for our experiments, i.e., a physical workstation (WS, Xeon X5680 3.3 GHz CPU, 12 physical cores, 140 GB RAM), and a VMware virtual machine (VM, Xeon E5-2650 2.60 GHz CPU, 16 physical cores, 128 GB RAM) on a cloud computing system provided by the University of Tokyo. We simply express these two environments WS and VM throughout the paper.

3.2.1 *ISoV* results

We implemented *ISoV* in a MiniSat 2.2, a SAT solver. We used a MiniSat solver, because this solver is the most representative sequential SAT solver. As MiniSat is a base solver for many other SAT solvers [94][44][95], the experimental results in this section can also be applied to many other solvers. Three hundred industrial instances from the SAT Competition 2014 were chosen as benchmarks. The execution environment was WS. Each instance was measured 12 times (1 MiniSat + *ISoV* 11 times). We can assume a hypothetical parallel SAT solver with 12 workers by gathering bests from 12 results. The limit time for each execution was set to 3600 s.

Figure 3.2 compares the results between MiniSat and *ISoV* on each instance. Each point on X-axis corresponds each instance, and the Y-axis represents the execution time. Each y value on the *ISoV* curve are gathering the best time for each instance from 11 *ISoV* results, since we want to compare the execution time between the original MiniSat and *ISoV*. *ISoV* performed better than MiniSat in many instances, but it also exhibited worse times in several instances. As *ISoV* simply shuffles indexs of variables, it is reasonable to consider 12 cases are equivalent in their average performances. This assumption indicates that any of the 12 cases could be the best solution in each instance. The curves in Figure 3.2 also indicate that a solver is diversified by *ISoV* at the extent when we observe their variations of execution times. Totally, a MiniSat solved 162 instances, and *ISoV* solved 183 problems with an additional 21 problems which were unsolved by MiniSat. The solving time was reduced in 105 instances, and 117 instances were remained unsolved. Table 3.1 shows the list of additinally solved 21 instances. The third row indicates the solved numbers from 11 *ISoV* tests. This results indicates that *ISoV* can improve several categories of instances, and is also applicable for both satisfiable and unsatisfiable problems, even though the number of unsatisfiable problems was small.

Figure 3.3 compares the MiniSat and the best time from 12 results (MiniSat + *ISoV*). This means each y value on the MiniSat + *ISoV* curve represents the time needed to solve an instance if they were ran in parallel. Thus, this figure shows the performance gaps between a MiniSat and a hypothetical parallel SAT solver with 12 workers. Therefore, the time amount as indicated by the space between the 2 curves will be improved by parallelizing a solver through *ISoV*.

In Figure 3.4, we compare the best, the worst, and the average time for each instance. We sorted data in ascending order along the average time. When the average time is infinity, the data is sorted in ascending order along how many times it reached the time-out limit. Variance seems very high when comparing the difference between the best time and worst time for each instance. In some instances, while the worst time reaches the time-out limit, the best time takes only a few seconds. Figure 3.5 shows the detailed execution times through *ISoV* for 2 instances. The execution times fluctuated dramatically between 40 seconds and the time limit in (a), and 2 seconds to 770 seconds in (b). At the very least, we can assume the execution time would be extremely changed for some types of SAT problems when we use *ISoV*.

Figure 3.6 shows a hypothetical parallel solver with *ISoV*s. For the curve named n workers, we gather the best times for each problem from the results of original MiniSat and *ISoV*s between seed value 1 and n - 1. A curve named 1 worker indicates the original MiniSat. There would be the performance limitation about *ISoV*, however based on our experiments *ISoV* showed its scalability at least up to 12 workers. To observe the limitation, more experiments would be needed using other seed values. However, measuring limitations through the scalability experiments might be unappropriate, because the

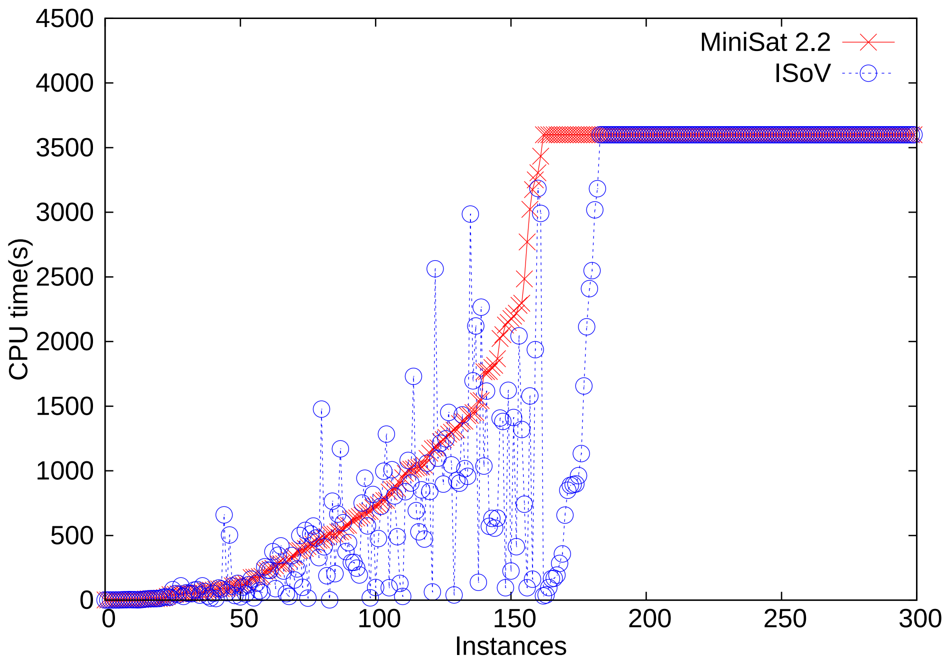


Fig. 3.2: Comparison of MiniSat and *ISoV*

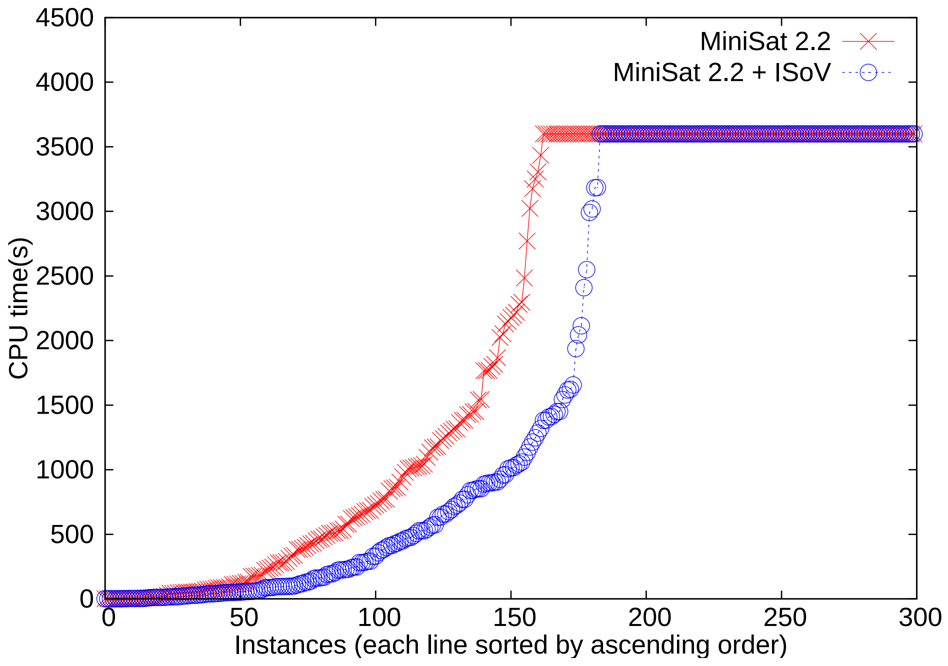


Fig. 3.3: Comparison of MiniSat and *ISoV*: Sorted by CPU time

Table 3.1: Instances unsolved by Minisat and solved by *ISoV*

Instance	Best Time (s)	Solved number	answer
002-80-8	3182.34	1	satisfiable
004-22-144	657.237	6	satisfiable
004-80-8	193.707	3	satisfiable
006-22-144	887.628	6	satisfiable
006-22-160	39.844	6	satisfiable
007-80-8	357.032	5	satisfiable
sat_dat.k30-30_rule	1656.43	1	satisfiable
UR-20-10p1	3018.72	1	satisfiable
UTI-20-10p1	2409.57	1	satisfiable
aaai10-planning-ipc5-pathways...	962.4	6	satisfiable
atco_enc1_opt1_04_32	32.9141	5	satisfiable
atco_enc1_opt2_10_16	1133.32	3	satisfiable
atco_enc2_opt2_10_21	164.598	8	satisfiable
gss-20-s100	283.93	5	satisfiable
gss-22-s100	96.654	3	satisfiable
gss-23-s100	849.297	1	satisfiable
hwmcc10-timeframe-expansion-k50...	2113.87	4	unsatisfiable
korf-18	897.892	7	unsatisfiable
openstacks-sequencedstrips...	2548.5	1	unsatisfiable
stable-400-0.1-5-9876543214005	892.992	1	satisfiable
vmpc_32.renamed-as.sat05-1919	169.103	6	satisfiable

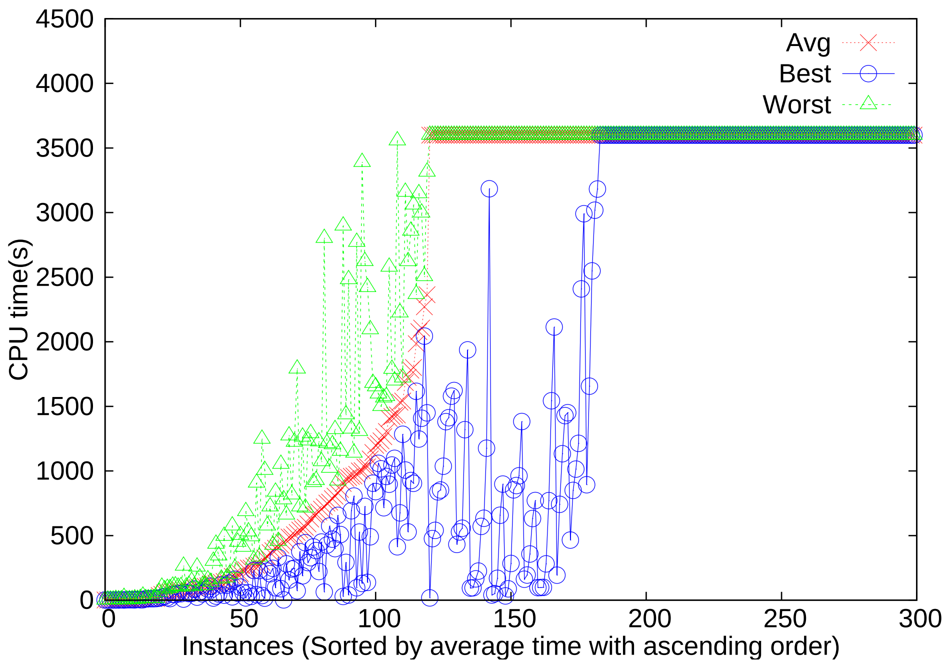
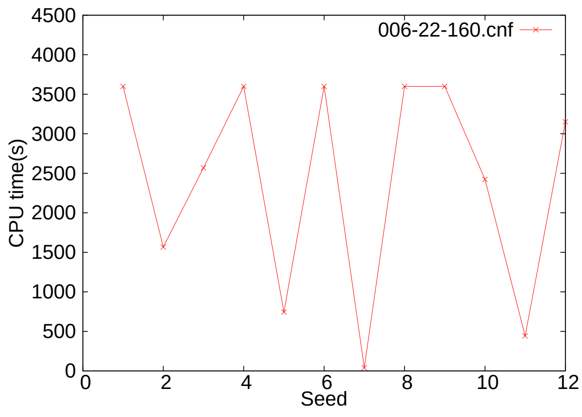
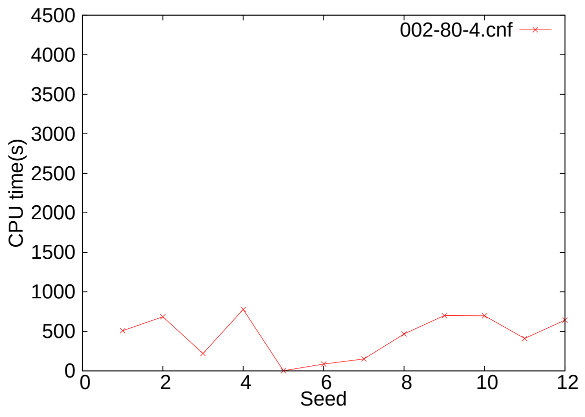


Fig. 3.4: Comparison of worst, best and average cases of *ISoV*



(a)



(b)

Fig. 3.5: Execution time changes through the changes of *ISoV*s

convergence of curves doesn't indicate the limitation of diversification. Clear thing we can say here is that, *ISoV* can produce search diversification with many different ways, because there exist $var!$ number of different orders through *ISoV*, when var is the number of variables in a SAT formula.

Figure 3.7 compares results with the different seed values. Best number for each seed indicates the number of instances those are solved the fastest using this seed number from 12 results. If we compare only the numbers, it seems to look that applying *ISoV* deteriorates the search efficiency, and it is against our assumption. However, when we compare their average running times, the differences between MiniSat and *ISoV* are not that significant. The results imply that *ISoV* definitely gives some kind of bad effects on search, however the extent is very small, and if we consider the extra diversifications produced by *ISoV*, this method would be appropiate for parallel SAT solvers.

3.2.2 Comparison of *ISoV* and *tie-breaking*

We assumpted in Section 3.1 that the impact of *ISoV* might be similar to the impact of the random selection from ties in the branching heuristics. For the comparison between *ISoV* and random *tie-breaking*, we implemented the random *tie-breaking* in MiniSat 2.2. MiniSat uses the VSIDS [14] as its branching heuristic, and the VSIDS in MinSat 2.2 is managed through the heap array structure. We implemented very naive implementation for the random *tie-breaking* by picking all the ties from the heap array, selecting one randomly,

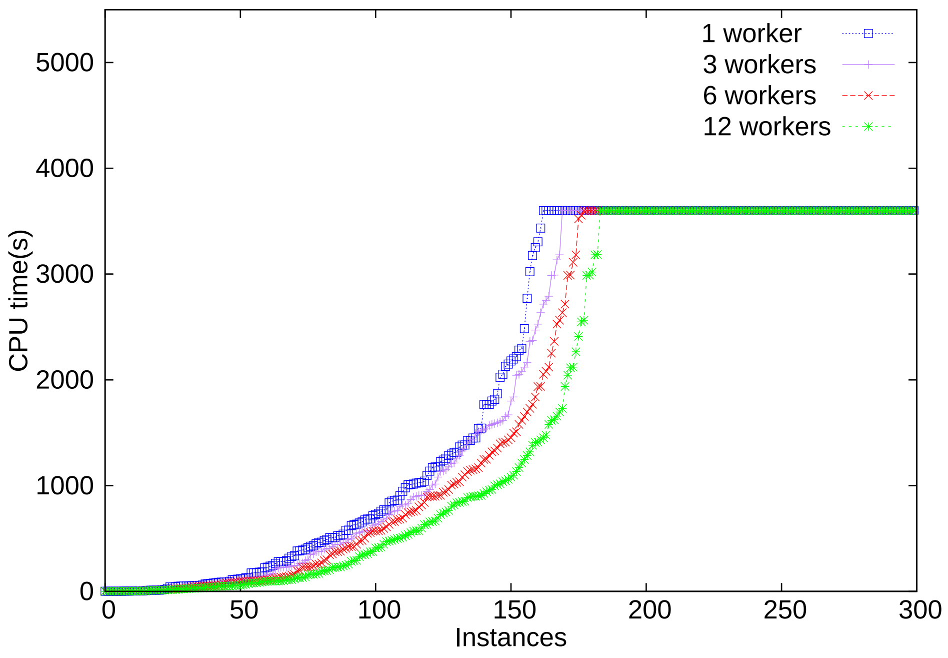


Fig. 3.6: Hypothetical parallel solver using worker with diffrent *ISoV*s

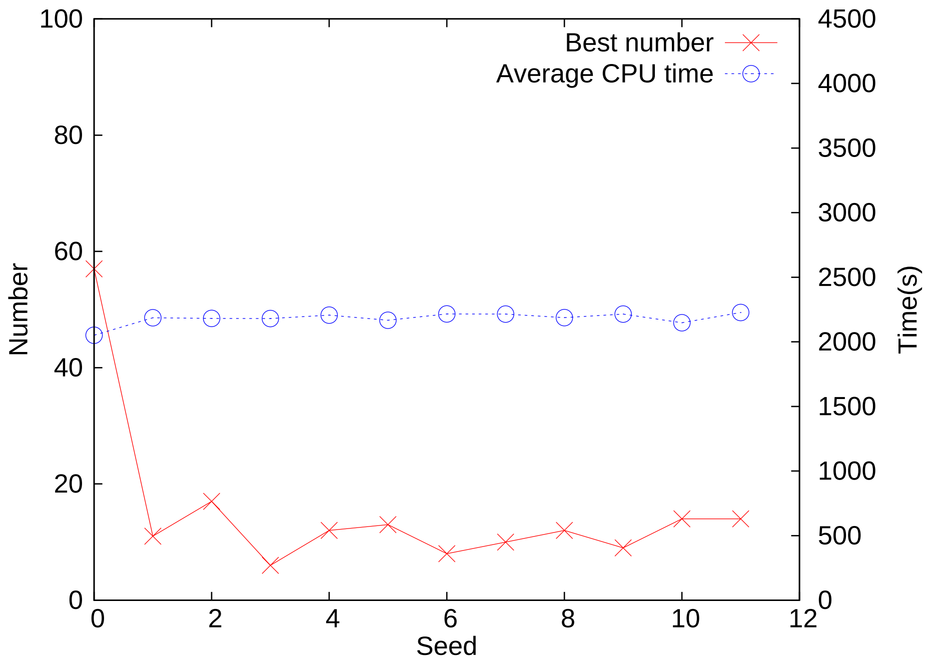


Fig. 3.7: Best numbers and average time of 12 (1 MiniSat + 11 *ISoV*) tests. Seed number 0 indicates results of the original MiniSat.

and pushing the rest of them into the heap array. We may be able to implement smarter by traversing from the root to obtain the ties list, and updating an array partial under the node of the randomly picked variable. However, our objective is the observation of similarities between *ISoV* and *tie-breaking*. Smart implementation is not needed here. Normally, VSIDS in MiniSat 2.2 works by picking a variable located at a root in the heap array. When there exist ties, pick all the ties recursively from the root, select one from them randomly, and insert the rest of them again in a heap array. This method needs time of O (2 $\times$ ties number $\times$ log (heap_size)) for each time we pick a variable in a branching heuristic. We know this is not smart, but enough to compare the tendencies between *ISoV* and *tie-breaking*.

We used the instance set of 21 industrial benchmarks shown in Table 3.1 which are additionally solved by using *ISoV* 11 times, and unsolved in the orginal MiniSat. We performed 10 *ISoV* tests and 10 *tie-breaking* tests for each instance. We performed *ISoV* again through different random seeds, because these 21 problems were already solved through the seeds value 1 to 11. Thus, using results of seed value 1 to 11 would be unfair for *tie-breaking*. Figure 3.8 shows the comparison between *ISoV* and *tie-breaking*. Each point indicates the best time from 10 tests. We calculated Pearson correlation coefficient. The value was 0.627076. This indicates *ISoV* and *tie-breaking* might have a strongly positive correlation at least w.r.t. the running time. We had only tested 10 times, and if we increase the number of tests, the value of correlation coefficient would be increased. In Figure 3.8, *ISoV* failed to solve 3 instances using random seed 10 times. This implies that we additionally solved 21 instances using *ISoV* in Section 3.2.1, but other instances can be additionally solved if we use different seeds. And this means *ISoV* would be a good method for parallel solvers to diversify search.

3.2.3 Comparison of *ISoV* and other random methods

We compared *ISoV* with other random methods, because we assumpted *ISoV* is useful because it produces extra diversity while maintaining intensity of search and would show better results compared with other random methods. We used 2 of existing options in MiniSat 2.2, those are *opt_random_var_freq* and *opt_rnd_init_act*. An option *opt_random_var_freq* indicates the frequency of the selections of random variables in the branching heuristic. In

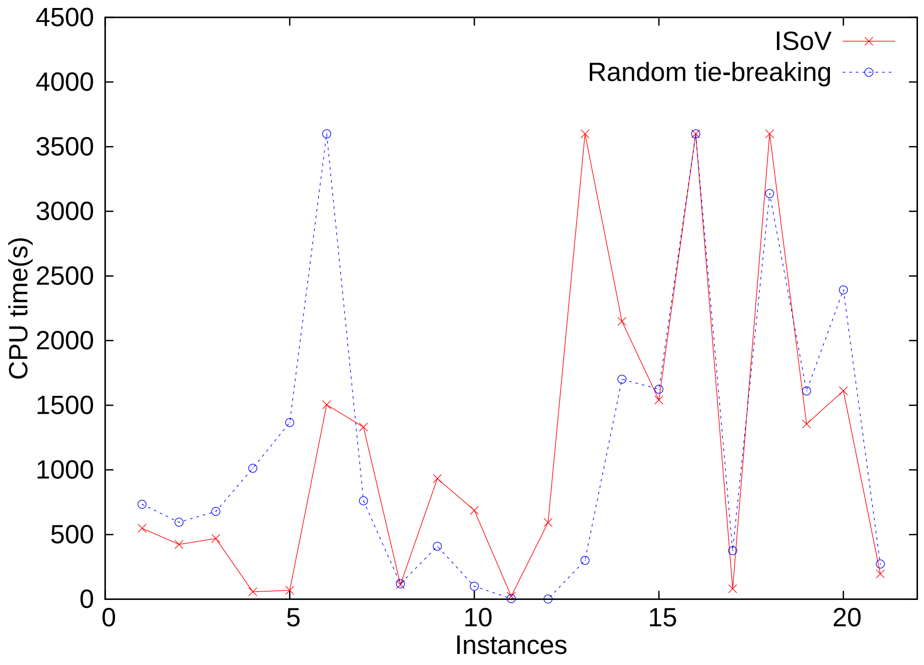


Fig. 3.8: Best time comparison of *ISoV* and *tie-breaking*. Each tested 10 times.

an initial version of Minisat 2, *opt_random_var_freq* was set to 0.02, and a random variable was chosen with a probability of 0.02. In Minisat 2.2, this option is not used because developers of Minisat 2.2 considered a random selection make the search inefficient. An option *opt_rnd_init_act* randomizes the initial *activity* which is used for scores in a branching heuristic. Each score is initialized by $d \times 0.00001$ ($0 \leq d < 1$). Thus, this option tries to ramdomized the several initial selections of variables in a branching heuristic.

Figure 3.9 compares among the *ISoV* and 2 randomization options stated above. A curve with a name *ISoV 1* is the results from Section 3.2.1 with seed values from 1 to 11, and a curve of *ISoV 2* is the results from Section 3.2.2 using random seeds, and Randomized params indicates results with the activations of 2 randomization options. We performed 10 tests for each instance. For each test numbered i ($1 \leq i \leq 10$), we set *opt_random_var_freq* to $0.01 \times i$ and set a different seed value to change the initialization of *activity*. Each curve is sorted by their running time. Each point indicates the best time from repeat tests. We can see that curves of *ISoV 1* and *ISoV 2* look very similar. This indicates the *ISoV*s with different seed values are showing the similar performances. Gaps between the curve of Randomized params and that of *ISoV*s are quite large. As we already showed in Figure 3.7, ISoV almost does not deteriorate the search efficiency. Therefore,

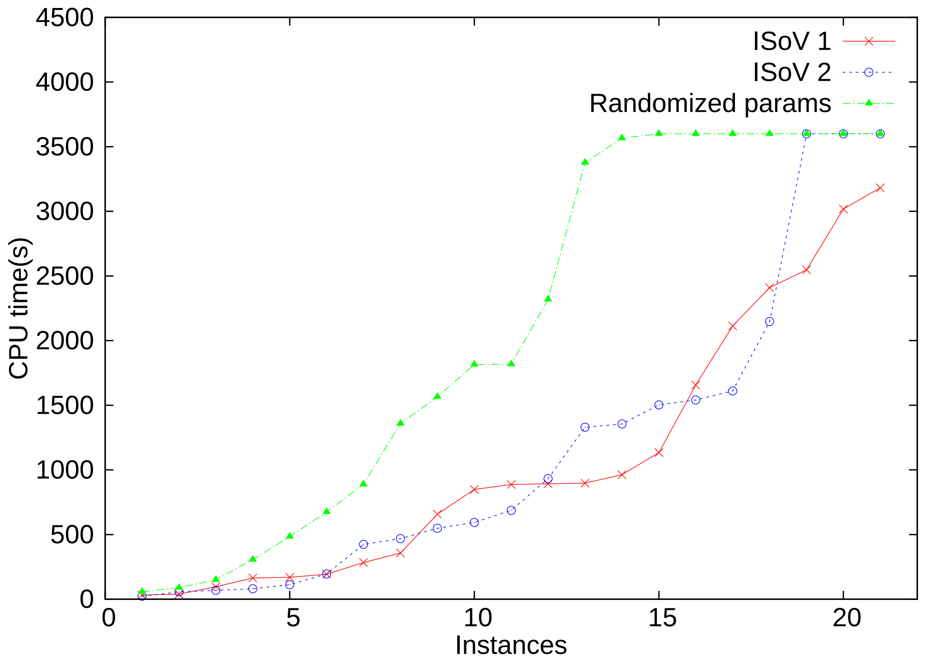


Fig. 3.9: Best time comparison of *ISoV* and *random methods*

we can conclude that *ISoV* showed the better performances than randomization options, and activation of these options largely deteriorated the search efficiencies by destroying the intensive search.

3.2.4 Experiments with *ISoV* on a parallel solver

We tested *ISoV* on MiniSat 2.2 in sections above, and compared its performances with random *tie-breaking*. For a while, we had tried to improve *tie-breaking* on MiniSat, and details will be discussed in Chapter 4. Three-hundred instances from application track in SAT Competition 2014 was too large for us to test repeatedly, because the performance evaluations using 300 instances required a lot of time. Therefore, we had picked a 49 instances set of MED49, MED means medium level of difficulty. Each instance in MED49 was solved at least once, and was unsolved at least 2 times from experiments in Section 3.2.1. While improving, we had doubts about the impact of *ISoV* in parallel solvers. Firstly, if we implement *ISoV* in a parallel solver, each worker have different index orders. Sharing learned clauses among workers is the essential for parallel solvers to avoid duplicate search and increase their performances. If we want to share learned clauses among

	Worker 1	Worker 2
Variables	$\{x_1, x_2, x_3, x_4, x_5, x_6\}$	$\{x_1, x_2, x_3, x_4, x_5, x_6\}$
Shuffle table	$\{x_3, x_4, x_2, x_5, x_6, x_1\}$	$\{x_5, x_6, x_4, x_3, x_1, x_2\}$
Restore table	$\{x_6, x_3, x_1, x_2, x_4, x_5\}$	$\{x_5, x_6, x_4, x_3, x_1, x_2\}$

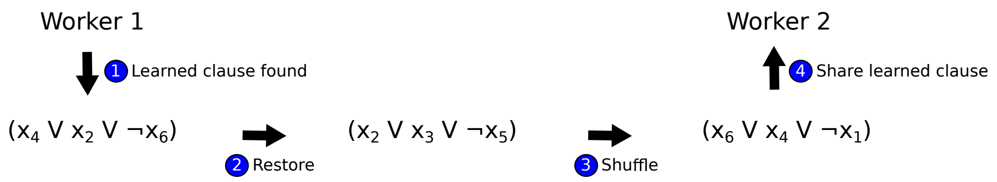

Worker 1
① Learned clause found
$(x_4 \vee x_2 \vee \neg x_6)$
② Restore
$(x_2 \vee x_3 \vee \neg x_5)$
③ Shuffle
Worker 2
④ Share learned clause
$(x_6 \vee x_4 \vee \neg x_1)$

Fig. 3.10: An example of a learned clause sharing in a parallel SAT solver with *ISoV*.

workers and each worker uses *ISoV*, restoration and shuffling processes would be required. Figure 3.10 shows an example of a learned clause sharing process. This example shows the flow of a learned clause sharing from a Worker 1 to a Worker 2. Each worker have both the shuffle table and the restore table respectively. Sharing process can be performed as follows.

① When a learned clause is found in a Worker 1, this one is already shuffled by a shuffle table in a Worker 1.

② A Worker 2 does not know information of tables in a Worker 1, thus a clause should be restored in a Worker 1 before the sharing.

③ To use a clause in a Worker 2, this one should be converted through a shuffle table in Worker 2.

④ After shuffling, a clause can be added to the list of learned clauses in a Worker 2.

Thus, if we use *ISoV* in the parallel solvers, the process of clause sharings needs additional restoration and shuffling sequences. This is why we actually implemented a parallel SAT solver to check if *ISoV* deteriorate its performances.

Secondly, most of the state-of-the-art parallel SAT solvers are using well-considered portfolios for efficient search. And as we mentioned above, they share learned clauses among workers. This is our second concern. We cannot assure that *ISoV* works well with clause sharings. We do not want to optimize different policies for different workers. Optimizing them requires a lot of efforts, and optimal portfolios would be changed based

Table 3.2: Experimental results of Glueminisat and ParaGluminisat. Columns except for Glueminisat are results on ParaGluminisat with different options.

Solver Type	satisfiable		unsatisfiable		Time (s)
	Solved	Time (s)	Solved	Time (s)	
Glueminisat	22	1096	9	535	2435
ISoV	26.6	1319	10	1146	2212
clause sharing	27.6	791	10	943	1798
ISoV + clause sharing	28.3	1105	10	876	1921
tie-breaking	26.6	1355	10	1253	2425

on the number of workers. Therefore, we adopted to used *ISoV* instead to generate search diversification. However, we have to check its validity with clause sharings.

We implemented our parallel solver ParaGluminisat. Glueminisat [95] is used as the base solver. Glueminisat is a solver based on MiniSat 2.2. It uses the LBD [44] measure for evaluation of learned clauses. Details of ParaGluminisat will be discuessed in Chapter 6. We used Glueminisat as a base solver instead of MiniSat because this solver is not a state-of-the-art solver anymore. Glueminisat is one of the best solvers, because it was the winner of application track in the SAT Competition 2011, and this solver adopted a fast restart policy based on the average of LBDs for recently learned clauses. Fast restart policy may produce enough diversity, and the impact of *ISoV* would be reduced. This is why we used Glueminisat, and if *ISoV* proves its diversity in ParaGluminisat, we might say that *ISoV* can be used as an option for state-of-the-art SAT solvers.

We compared Glueminisat and ParaGluminisat with several options for MED49. Options we implemented are *ISoV*, *clause sharing* and *tie-breaking.* Table 3.2 shows the summary of ParaGluminisat results. The number of workers was 12. Glueminisat is tested 1 time, and tests of ParaGluminisat with diffrents options were performed 3 times and averaged, because results of parallel SAT solvers have larger run time variations than those of sequential solvers especially for satisfiable problems. We divided solved problems into satisfiable and unsatisfiable instances and showed solved instances and the average time respectively. Average times in column 3 and 5 are calculated only for solved instances. Average times in column 6 are calculated by all instances. For unsolved instances we add 5,000 s which is the number for time limit. Solvers with *clause sharing* option shared clauses within $LBD = 2$.

All results with parallel solvers showed better performance than those of a sequential solver Glueminisat. Results of *ISoV* and *tie-breaking* are very similar, but average time in *ISoV* was faster than those of *tie-breaking* because as we abovementioned our *tie-breaking* method here is implemented with a naive approach, thus it required a lot of time to pick a tie. When we compare the average time we have to compare them considerately. If we only compare the times of Glueminisat and *clause-sharing* for unsatisfiable instances, Glueminisat looks faster than *clause-sharing*. However, this is because *clause-sharing* additionally solved one instance than Glueminisat, and run time of that instance was over 4,000 s. Even though we consider the factor that additionally solved instances usually take a lot of time, there exists trade-off when we apply *ISoV* and *clause sharing*. By applying these options, we could solve hard problems compared to Glueminisat, but run times for easy problems became longer.

When we compare the results between *ISoV* and *ISoV* + *clause sharing*, we can observe *clause sharing* make search better by combining them. However, when we compare the results between *clause sharing* and *ISoV* + *clause sharing*, it is difficult to determine which one showed the better results. Even though *ISoV* + *clause sharing* solved slightly more problems than *clause sharing*, the difference is too small, and average run time became longer.

For the results of unsatisfiable instances, the number of solved instances are the same regardless of their options. However, if we scuritiny details of them, list of them were little bit different. Figure 3.11 shows the run times of 11 unsatisfiable problems. For each option, we tested 3 times. Figure 3.11 (a), (b), and (c) show results of each *ISoV*, *clause sharing*, and *ISoV* + *clause sharing*. These results indicate that the run times of parallel SAT solver are quite stable when they solve the unsatisfiable problems. Figure 3.11 (d) compares these three options. Curves of *ISoV* and *ISoV* + *clause sharing* shows that *clause sharing* option make search efficiency better when we use *ISoV*. If we compare curves between *ISoV* + *clause sharing* and *clause sharing*, *ISoV* make search efficiency worse in the most of instances. However, we want to focus on instances numbered 7 and 8. Instance 7, named "korf-18.cnf", indicates a rectangle packing problem [96]. Instance 8, named "openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_1.025-NOTKNOWN.cnf", is included in planning category. We call an instance 7 to korf, and

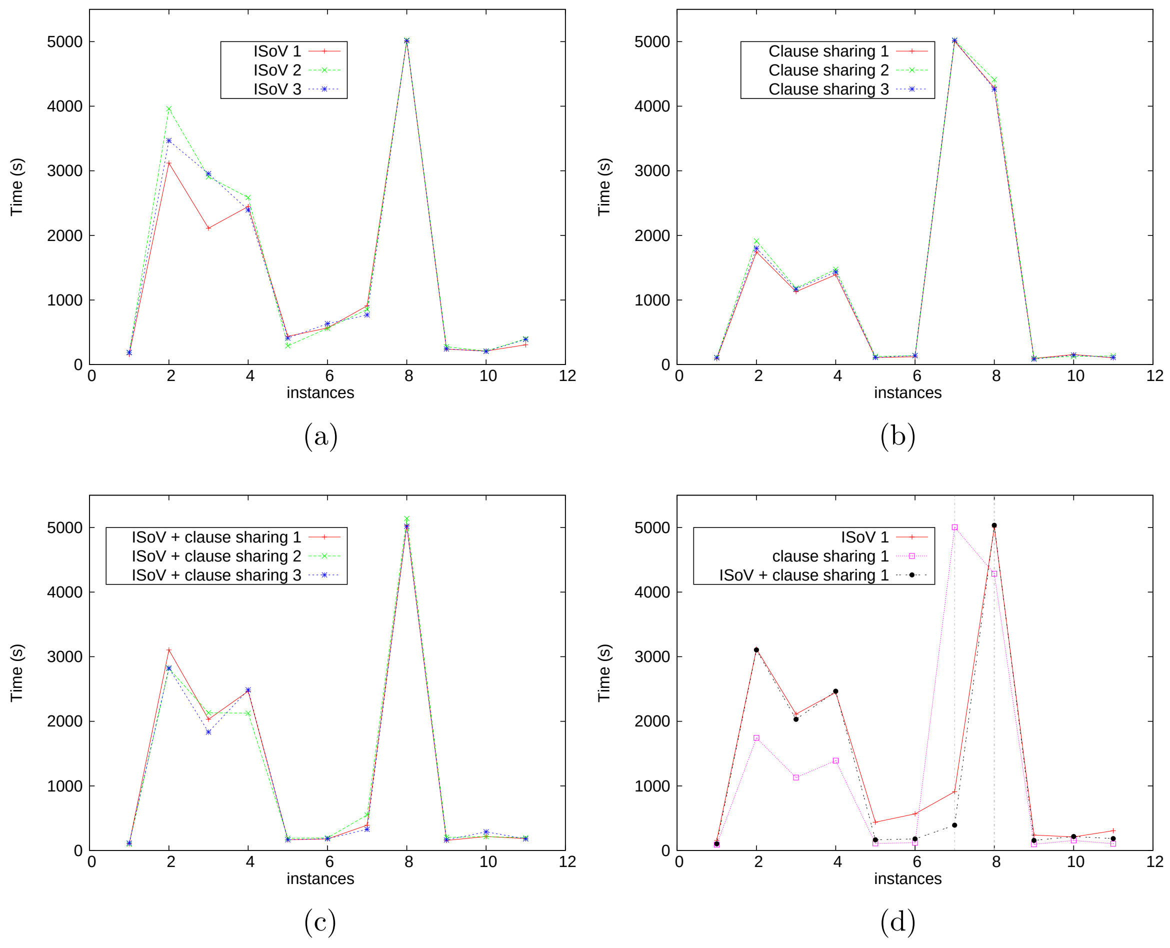


Fig. 3.11: Run time comparison of ParaGluminisat with different options.

call an instance 8 to openstacks. For korf, *ISoV* option worked well. For openstacks, a solver with *clause sharing* option solved this at 4,319 s on average. Time limit was set to 5,000 s, and results with *ISoV* option reached at the time limit. When we compare performances among different solvers penalized average runtime (PAR) [97] is widely used. When we use PAR10 score, it gives penalty to unsolved problems 10 $\times$ T, when T is a time limit. If we evaluate *ISoV* + *clause sharing* and *clause sharing* using PAR10 score, *clause sharing* will be assessed a better solution. However, we considered that *ISoV* + *clause sharing* might solve openstacks when we extend time limit a little bit. This is because, in the most of cases, *ISoV* + *clause sharing* needed more of time than *clause sharing*. On the other hands, we cannot assume how long *clause sharing* takes to solve korf because when we look other instances *clause sharing* solved faster than *ISoV*, but

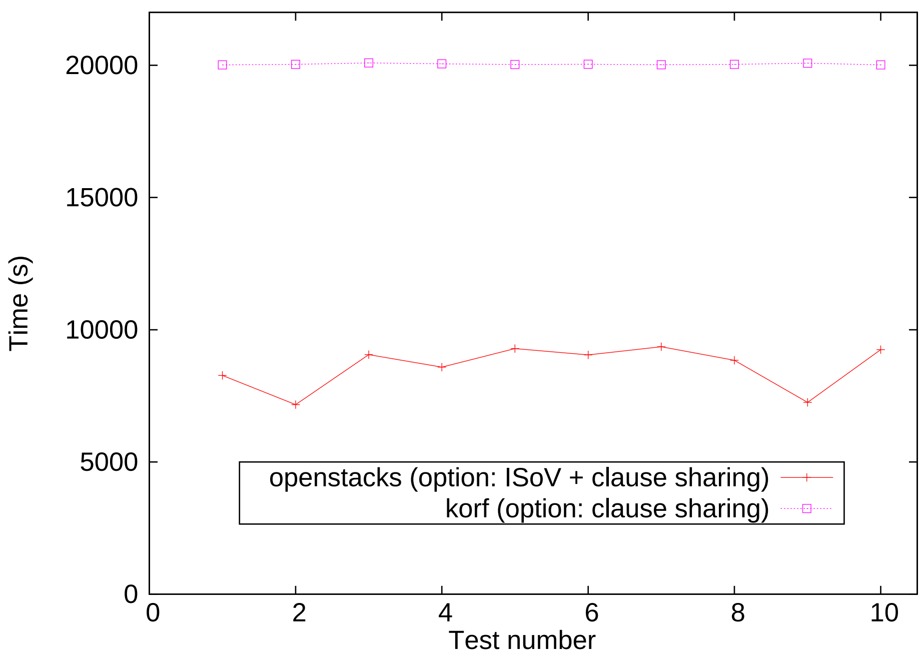


Fig. 3.12: Results of two instances those are unsolved with option of either *ISoV* and *clause sharing*

aspect for korf is different. Thus, we extended time limit to 20,000 s and tested 10 times. Figure 3.12 shows results of korf and openstacks. As we expected, *ISoV* + *clause sharing* solved openstacks under 10,000 s all the time. However, *clause sharing* failed to solve korf in 20,000 s all the time.

We only concentrated on two instances, but we comfirmed the effect of *ISoV* is not a subset of those of *clause sharing*. Finally, we further extended time limit to 100,000 s to solve korf. Then a solver with *clause sharing* solved korf and it took 77,757 s. This result is 20 times slower than a solver with *ISoV* + *clause sharing*. Therefore we concluded *ISoV* could make diversity to reduce falling into the desert, and we adopted this as one of default features of our parallel SAT solver ParaGluminisat.

Chapter 4

New branching heuristic

We have observed small changes in a branching heuristic may change the performance of a SAT solver a lot in Chapter 3. In Section 3.2.2, we performed random *tie-breaking* in a branching heuristic and learned several instances can be solved through the *tie-breaking* method. These results represent the potential of *tie-breaking* to improve the performance of branching heuristics.

In this chaper, we propose a method to break ties in the branching heuristics. We evaluated *tie-breaking* through the two branching heuristics, VSIDS and CHB. We mainly discuss about *tie-breaking* of VSIDS (TBVSIDS). The initial VSIDS was proposed in 2001 [14], and we call its modern version, EVSIDS, to VSIDS as forementioned in Chapter 2. There were lots of efforts to outperform this branching heuristic, but VSIDS is widely being used in a lot of SAT solvers for longer than 10 years because of its robustness. We also discuss about *tie-breaking* of CHB (TBCHB) briefly. Two branching heuristics are utilized to evaluate *tie-breaking*, but we consider this method is also applicable to many other branching heuristics.

We briefly address the structure of this chapter. Firstly, we observe how frequently ties occur in VSIDS. Secondly, we describe *tie-breaking* for VSIDS and CHB, respectively. Several methods including initial approaches for TBVSIDS are introduced, and TBCHB is described. Finally, we show the results of TBVSIDS and TBCHB, and we analyze the performance of TBVSIDS by comparing several factor with those of VSIDS. All of our methods are evaluated through the benchmarks from the SAT Competitions.

4.1 Observation of tie occurrences

Our objective in this section is the observation of tie frequency in a branching heuristic. We observe tie occurrences to ensure the necessity of efficient *tie-breaking* method. If ties occur frequently, an appropriate *tie-breaking* method might lead the better search efficiencies.

Our observation here is performed through Glucose 3.0. Glucose adopts VSIDS as its branching heuristic, and many of modern SAT solvers also adopt VSIDS. Therefore, solvers with the same branching heuristics may show the similar results described in this section. Benchmarks chosen for the tie observations was 300 instances from SAT Competition 2014 application track. Time limitation was set 1,000 s for each instance. Figure 4.1 shows the number of decisions, ties and tie/decision ratios for each instance. Results are sorted by tie ratios in ascending order. Instances solved within 1,000 s were excluded, because they are too short for the calculations of tie ratios. If an observation time is too short, the tie ratio might be observed very high, because all scores of variables are initialized by zero initially.

We observed each decision and checked whether ties exist in a branching heuristic. Ties and decisions were shown using the logarithm scale for an object to display them in a graph, because their values were ranged from 2 million to 200 million. We can observe from this Figure that, ties occur very frequently for some instances. The median ratio was 0.05, and the mean ratio was 0.19. Thus, ties occur once in five conflicts on average. These results indicates that it might be worth designing a tie-breaking policy at least if we use VSIDS as a branching heuristic.

We can presume with several hypotheses that why ties occur frequently in a branching heuristic, especially in a VSIDS. If a conflict is induced from a lot of clauses, and the variables in them are dissimilar, then a lot of variables are incremented by 1 simultaneously. If several clauses are used as reasons of conflicts continuously, variables in them would obtain a lot of scores, and they may produce and maintain ties in a branching heuristic in a while. We can also consider other scenarios. Consider the situation that the list of variables related to a conflict changes a lot, i.e, the area of search space changes and search diversify is very high. In this case, a lot of variables may obtain their scores and their

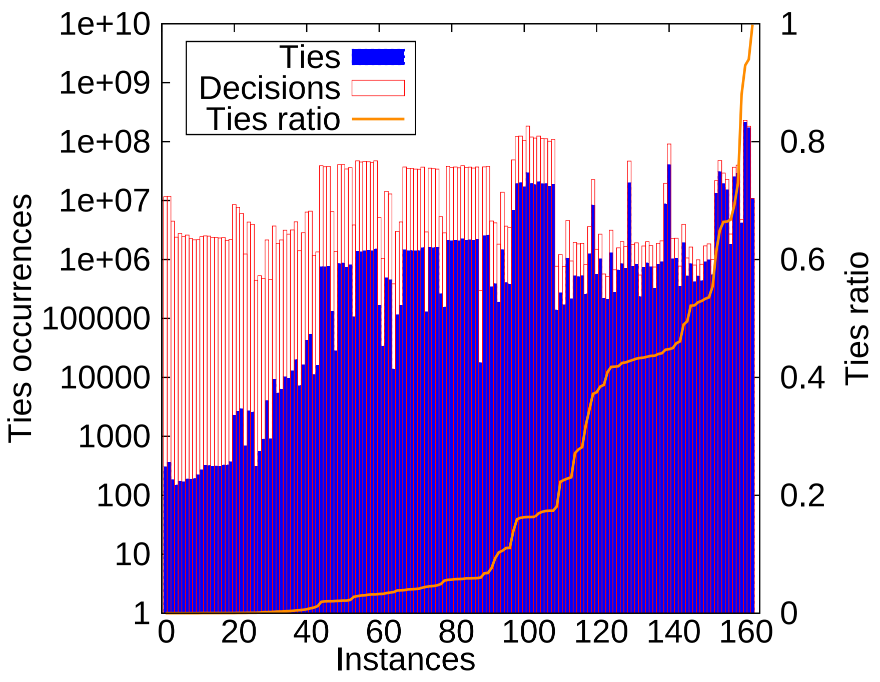


Fig. 4.1: Tie occurrences of instances from SAT Competition 2014 application track.

scores would converge to several numbers. We can also hypothesize many other scenarios for tie occurrences, but further scrutinization of ties in VSIDS or distributions of scores in VSIDS might give informative intuitions [98] for improving their policies.

4.2 Details of *tie-breaking*

In this section, we describe details of *tie-breaking* on two branching heuristics, VSIDS and CHB with their pseudo codes.

4.2.1 Initial approaches for TBVSIDS

We introduce our initial *tie-breaking* method here. Our initial approach is more complicated than our later approaches. Although we are currently not using this version for *tie-breaking*, we describe details here because parts of the initial approaches might be handy for designing the efficient branching heuristic. Indeed, we adopted several parts from the initial approaches into our latest version for *tie-breaking*. Therefore, we display our initial approaches here and show the details of pseudo code in algorithm 2. Note that details of the implementation are little bit distant from the pseudo code described here.

We determined to admit and maintain the power of VSIDS as far as possible, because VSIDS had been demonstrated its efficiency through the thousands of benchmarks for more than ten years. As an attempt to keep the efficiency of VSIDS, we designed the concept of *tie-breaking.* A variable with the highest VSIDS score is always selected on all decisions.

To achieve *tie-breaking* from ties, we considered 2 indicators to differentiate them. We describe a pseudo code in Algorithm 2. This one is designed to break ties in VSIDS. Scores of variables in VSIDS are managed in *activity* array, and the order of them is managed by the heap array. Firstly, we prepared another array and named it $activityMini$ (line 3). The $activityMini$ is updated when a clause is obtained from a conflict analysis or removed from the clauses database (line 12 and 40). The *dist*(*learned clause*) indicates the size of a learned clause, i.e., the number of literals in a clause (line 11). Secondly, we considered a function w for the varables in ties (line 24). Each variable v corresponds to 2 literals v and $\neg v$, and $w(v)$ is obtained by the sum of *watch* lengthes of v and $\neg v$. Each literal has a *watch* list, i.e., a list of clauses. The structure of them is related to two-watched literals tequnique explained in Section 2.3. Each of learned clause has two watched literals in it, and each of watched literals has a *watches* of clauses accordingly.

An example of *watches* can be described in Figure 2.3. For a clause $(x_1, x_2, x_3, x_4, x_5)$, it has two watched literals of x_1 and x_5. This clause is included in watches of $\neg x_1$ and $\neg x_5$ initially, not x_1 and x_5 because we want to watch falsified points. Therefore, when a *FALSE* value is assigned to x_1, we update *watches* of $\neg x_1$. As there is an update of a watched literal in step 3, a clause is removed from *watches* of $\neg x_5$ and added to *watches* of $\neg x_2$.

Initially, we designed a little complicated approach for assessing $w(v)$. We wanted to prioritize a variable if it is related to a lot of short clauses such as the binary/ternary clauses. However, the two-watched literal technique maintains a lazy data structure and there is no order relation between two-watched literals. Therefore, we attempted to traverse all clauses related to the candidate variables, excluded satisfied clauses, and counted the number of unassigned variables for not satisfied clauses. We considered this approach might work, even though it traverses all clauses related to candidiate variables. For ex-

Algorithm 2 Initial *tie-breaking* branching heuristic.

```
for $v \in$ Vars do
    $activity[v] \leftarrow 0$
    $activityMini[v] \leftarrow 0$
end for
loop
    if a conflict occurs then
        for $v \in$ variables resolved during conflict analysis do
            $activity[v] \leftarrow activity[v] + 1$
        end for
        for $v \in learned\ clause$ do
            $quality(learned\ clause) \leftarrow 1\ /\ dist(learned\ clause)$
            $activityMini[v] \leftarrow activityMini[v] + quality(learned\ clause)$
        end for
    else
        $unassigned \leftarrow$ unassigned variables
        $actMax \leftarrow argmax_{v \in unassigned} activity[v]$
        $ties \leftarrow \emptyset$
        for $v1 \in unassigned$ do
            if $activity[v1] == actMax$ then
                $ties \leftarrow ties \cup \{v1\}$
            end if
        end for
        $actMiniMax \leftarrow argmax_{v' \in ties} activityMini[v']$
        $watchMax \leftarrow argmax_{v' \in ties} w(v')$
        $candidates \leftarrow \emptyset$
        for $t \in ties$ do
            if $actMiniMax \times th1 < actMini[t]$ && $watchMax \times th2 < w(t)$ then
                $candidates \leftarrow candidates \cup \{t\}$
            end if
        end for
        $PF \leftarrow$ Pareto front from $candidates$
        $v^* \leftarrow argmax_{v1 \in PF} maxP(v1)$
        return $v^*$
    end if
    if conflict number meets reduceDB condition then
        for $lc \in learned\ clauses$ do
            if Erase $lc$ then
                for $v \in lc$ do
                    $quality(lc) \leftarrow 1\ /\ dist(lc)$
                    $activityMini[v] \leftarrow activityMini[v]$ - $quality(lc)$
                end for
            end if
        end for
    end if
end loop
```

ample, there are a hundred thousand variables and m clauses in a SAT formula, and we can assume a hundred number of candidate variables in a branching heuristic, then only $0.001 \times m$ clauses would be traversed. However, we did not considered the decision rate. Decisions are performed around 2 thousand to 200 thousand times per second and traversing clauses on each decision is too costly. Therefore, the initial w made the propagation rate of a solver too late, and we had to simplify w by only considering lengths of $watches$s. We did not implemented in this time, but there might be a chance to speed up the initial w model by adding and updating a counter for each clause which represents the number of unassigned literals in a clause. Based on 2 indicators abovementioned, we calculated a Pareto front [99] in each decision and choose an appropriate one from them (line 31).

We considered there might be several problems to adopt initial aproaches for a branching heuristic. Firstly, an algorithm includes several of time consuming processes. A solver have to calculate a Pareto front directly on each decision in a branching heuristic. As the lengths of $watches$s through 2 watched literals change dynamically, a solver has to calculate a number of unassigned variables exactly when it needs. Further, number of ties turns quite big for sometimes, and considering all ties might require a lot of time (line 20). Secondly, an algorithm has several dimensions for the optimization. We set the limit number for ties (line 26) for avoiding the large amount of ties. A solver eliminates variables with low $activityMini$ scores or low lengthes of $watches$ from the ties through $th1$ and $th2$ (line 27). A solver picks a variable from a Pareto front (line 32), and currently it is simply picked from the front.

Even though the exsistence of problems abovementioned, several ideas and approaches in this algorithm showed quite promissing results which are remarked in Section 4.3.

4.2.2 New approaches for TBVSIDS

Experiments in our initial *tie-breaking* method using only *activityMini* showed the reasonable results in Section 4.2.1. Therefore, we decided to adopt this idea. Score orders of *activity* and *activityMini* are different. However, we can maintain variables by descending order by prioritizing *activity*, and breaking ties of *activity*s using *activityMini*. A solver does not need to compare all ties directly on each conflct by modifying the comparison function of scores between two variables. Instead of *activityMini*, a solver updates *activi-*

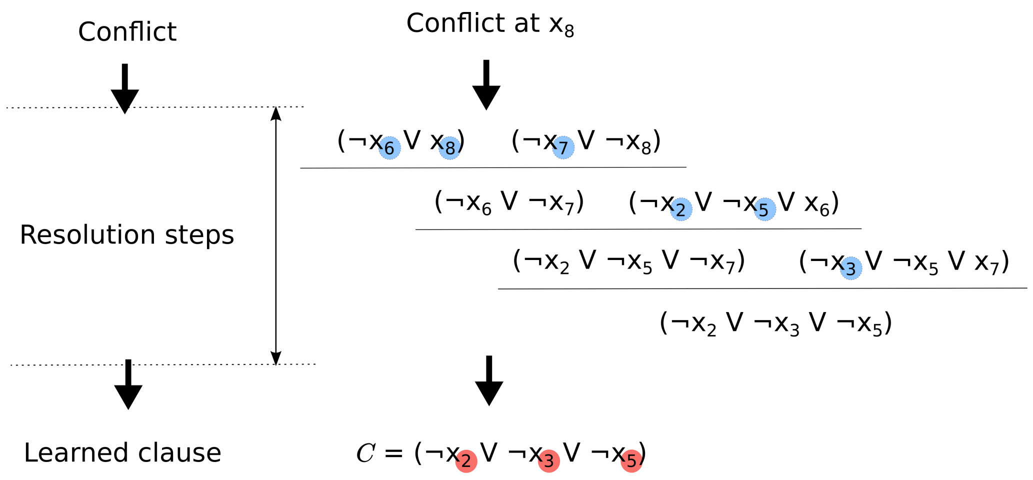

Conflict
Conflict at x_8
Resolution steps
$(\neg x_6 \vee x_8)$ $(\neg x_7 \vee \neg x_8)$
$(\neg x_6 \vee \neg x_7)$ $(\neg x_2 \vee \neg x_5 \vee x_6)$
$(\neg x_2 \vee \neg x_5 \vee \neg x_7)$ $(\neg x_3 \vee \neg x_5 \vee x_7)$
$(\neg x_2 \vee \neg x_3 \vee \neg x_5)$
Learned clause
$C = (\neg x_2 \vee \neg x_3 \vee \neg x_5)$

Fig. 4.2: Clause learing through conflict analysis.

*tyQ*s at different place with different ways. To achieve the better or at least the equivalent performance of a SAT solver, *tie-breaking* with *activityQ* has to select the better selections than those in the original VSIDS, and the simple and agile calculation method would be required for *activityQ*.

As forementioned in Chapter 2, VSIDS increments all scores of variables related to resolution steps. All the previous scores are automatically decayed, because we add the bumped scores for variables in resolution steps. By bumping and decaying, VSIDS emphasizes recent learnings for the intensive search. When we consider the objective of branching heuristics for achieving intensive search, *tie-breaking* for strengthening search intensification might improve the improvement of a SAT solver.

Figure 4.2 describes an example of the resolution steps. When a conflict occurs, resolution steps are performed from a conflict node for an object to learn a new clause. VSIDS increments all variables related to resolution steps without duplication. Increments are performed only the variables appeared at the first time in the resolution steps and displayed by blue circles in Figure 4.2.

We want to pay attention to variables in a learned clause displayed with red circles. In VSIDS, variables x_2, x_3, x_5, x_6, x_7 and, x_8 are bumped as the same way. All of these variables are related to occur the same conflict and considered equivalently in VSIDS. Considering all variables in the resolution steps equivalent might be reasonable, because all of them could be included in a learned clause. When a solver learns a clause throught a first-UIP cut, $C = (\neg x_2 \vee \neg x_3 \vee \neg x_5)$ would be obtained. However, we may also learn $C = (\neg x_6 \vee \neg x_7)$ or $C = (\neg x_2 \vee \neg x_5 \vee \neg x_7)$ instread.

However in our consideration, bumping all variables in a conflict side and a reason side with the same amount would be unfair. Through the resolution steps, we actually learns a cluase C. And C would be reused for the unit propagation or in a conflict analysis, when two of three literals in it are falsified. If C is not reused during search, a conflict analysis for C become a useless process. We considered we may reuse the learned clauses more actively by giving bonus scores to variables in the learned clauses.

Algorithm 3 TBVSIDS1 branching heuristic.

```
1: for $v \in$ Vars do
2:     $activity[v] \leftarrow 0$
3:     $activityQ[v] \leftarrow 0$
4: end for
5: loop
6:     if a conflict occurs then
7:         for $v \in$ variables resolved during conflict analysis do
8:             $activity[v] \leftarrow activity[v] + 1$
9:         end for
10:        for $v \in$ learned clause do
11:            $quality(learned\ clause) \leftarrow 1\ /\ dist(learned\ clause)$
12:            $activityQ[v] \leftarrow activityQ[v] + quality(learned\ clause)$
13:        end for
14:    else
15:        $unassigned \leftarrow$ unassigned variables
16:        $v' \leftarrow argmax_{v \in unassigned} activity[v]$
17:        if $v'$ in ties then
18:            $v^* \leftarrow argmax_{v \in ties} activityQ[v]$
19:        else
20:            $v^* \leftarrow v'$
21:        end if
22:        return $v^*$
23:    end if
24: end loop
```

Algorithm 3 shows the detailed pseudo code including our considerations abovementioned. We prepared *activityQ*, and this one works little bit different compared with *activityMini* in algorithm 2. There was a update of *activityMini* when a learned clause is attached or detached into the learned clauses database. However, *activityQ* is only updated when a learned clause is obtained, and scores in it are decayed with the same way for *activity* in VSIDS (line 12). This means we give bonus scores to variables in a learned clause, and the variables in a recently learned clause get bigger bonuses. Decision parts (line 14 - 22) become quite a simple compared with the initial algorithm. First, we pick a variable or variables with the highest *activity*, and break ties by picking a variable with the highest *activityQ* if ties exist. We are expecting the number of ties would be reduced by applying *tie-breaking*, but there can still occur when both *activity* and *activityQ* are equal among variables.

Algorithm 4 TBVSIDS2 branching heuristic.

```
1: for $v \in$ Vars do
2:     $activity[v] \leftarrow 0$
3: end for
4: loop
5:     if a conflict occurs then
6:         for $v \in$ variables resolved in conflict analysis do
7:             $activity[v] \leftarrow activity[v] + 1$
8:         end for
9:         for $v \in lc$ ($lc$: learned clause) do
10:            $quality(lc) \leftarrow 1 / (k \times dist(lc))$
11:            $activity[v] \leftarrow activity[v] + quality(lc)$
12:        end for
13:    else
14:        $unassigned \leftarrow$ unassigned variables
15:        $v^* \leftarrow argmax_{v \in unassigned} activity[v]$
16:        return $v^*$
17:    end if
18: end loop
```

Algorithm 4 describes another *tie-breaking* method. Algorithm 4 looks very similar to Algorithm 3. However there exist the big differences between them. In TBVSIDS2, a solver directly add bonuses to *activity* instread of preparing *activityQ*. This would change the meaning of *tie-breaking* a lot. Firstly, we considered direct bonuses to *activity* would change the performance of VSIDS more than TBVSIDS1. Persistent bonuses in *activity* would diversify scores in *acitivity*, and accumulation of them might be more influential than *activityQ* to break ties. Secondly, we considered *activity* does not consider the possibility of unit propagation in learned clauses, as we explained above through an example in Figure 4.2. If we give bonuses to *activity* directly but relatively smaller than increments in VSIDS, a solver might break ties while minimizing deformation of direct changes of VSIDS. There is a parameter k for *quality* calculation in Algorithm 4. When k value is set too high, TBVSIDS2 works such as VSIDS. If k is set too low, scores obtained by resolutions would be ignored. Therefore, adjusting k for the proper scaling of *quality*s in TBVSIDS2 is very important. As we would like to admire the performance of VSIDS, assigning a k value larger than 1 would be preferred.

About the *quality* of a learned clause, a solver uses either the size and the *LBD* indicator. The selections between them are considered to strengthen the interplays between a clause learning scheme and a branching heuristic. We discuss details about this at the experimental results in this chapter.

4.2.3 Details for TBCHB

We describe details of TBCHB in this section. CHB heuristic updates *activity*s on each decision, and VSIDS updates *activity*s on each conflict. Further, CHB updates *activity* of each variable v based on its reward computed by the inverse distance between current conflict number and the last conflict number that have a connection with v. Therefore, a variable recently used in a conflict gets more score. Now, scores of variables updated by a conflict are different on each variable basis. We attempted to apply Algorithm 4 to CHB, because we considered addtional scores for variables in a learned clause might handy for any branching heuristic to strengthen the interplay between a learning scheme and a branching heuristic. To achieve this, we need to adjust the *tie-breaking* method for CHB with the different ways compared with methods in VSIDS.

Algorithm 4 describes details of TBCHB2 branching heuristic. A conflict occurs after several decisions. We prepared a $maxD$ to obtain the maximum increase among score updates between the conflicts. This value is reset to zero, after a conflict occurs and *tie-breaking* is applied (line 19). We uses $maxD$ to adjust the scale of our *quality*s (line 16 - 17). We attempted to maintain the original order in CHB by adding relatively small scores through the calculation of $maxD$. Results of TBCHB is shown in the following section.

Algorithm 5 TBCHB2 branching heuristic.

```
for $v \in$ Vars do
    $activity[v] \leftarrow 0$
    $maxD \leftarrow 0$
end for
loop
    if a conflict occurs then
        for $v \in$ variables just propagated do
            A $\leftarrow activity[v]$
            $activity[v] \leftarrow activity[v] \times (1 - \alpha) + reward \times \alpha$
            B $\leftarrow activity[v]$
            if $maxD <$ B - A then
                $maxD \leftarrow$ B - A
            end if
        end for
        for $v \in lc$ ($lc$: learned clause) do
            $quality(lc) \leftarrow maxD \,/\, (k \times dist(lc))$
            $activity[v] \leftarrow activity[v] + quality(lc)$
        end for
        $maxD \leftarrow 0$
    else
        for $v \in$ variables just propagated do
            A $\leftarrow activity[v]$
            $activity[v] \leftarrow activity[v] \times (1 - \alpha) + reward \times \alpha$
            B $\leftarrow activity[v]$
            if $maxD <$ B - A then
                $maxD \leftarrow$ B - A
            end if
        end for
        $unassigned \leftarrow$ unassigned variables
        $v^* \leftarrow argmax_{v \in unassigned} activity[v]$
        return $v^*$
    end if
end loop
```

4.3 Experimental results

4.3.1 Initial approach of TBVSIDS

We show results of our initial approach introduced in Section 4.2.1. We implemented algorithm 2 into MiniSat 2.2. Time limit was set to 3,600 s and execution environments was WS.

A MED49 benchmark set is used for evaluation. Results are shown in Table 4.1. To observe the effectiveness of 2 indicators $activityMini$ and w, we implemented each side respectively, performed each and combined 2 indicators. Applying only *activityMini* worked well. However, activating only a function w did not increased solved instances. However, when we consider the time consumption for calculating ties, we might insist w lead search to good way. As results with combination of 2 indicators showed the best performances, our *tie-breaking* method with a Pareto front showed its efficiency at least for MED49. Further improvement would be done through algorithm configuration tools such as paramILS [97] or SMAC [100]. Applying dynamic programming for function w and proposing better policy to pick a variable from Pareto front would also improve this algorithm.

Table 4.1: Experimental results of initial *tie-breaking* on MiniSat 2.2.

Branching heuristic	Solved / Problems
VSIDS	28 / 49
VSIDS + *activityMini*	33 / 49
VSIDS + w	28 / 49
VSIDS + *activityMini* + w	35 / 49

4.3.2 Comparison among VSIDS and TBVSIDSs

We compare the performances among VSIDS and TBVSIDSs through benchmarks from SAT Competitions.

In our initial algorithm in Section 4.2.1, we used the size of a learned clause to break ties. First, we adopted this idea in the algorithms 3 and 4 and implemented TBVSIDSs in MiniSat 2.2. For an algorithm 4, k is set to 1 as a default. Table 4.2 and Figure 4.3 describes results on MiniSat 2.2. Time limit was set 3,600 s and used benchmarks were 300 application instances from SAT Competitions 2014.

Table 4.2 shows both TBVSIDS1 and TBVSIDS2 solved more instances than VSIDS.

Table 4.2: Experimental results of VSIDS and TBVSIDSs on MiniSat 2.2.

		VSIDS	TBVSIDS1	TBVSIDS2
2014App	satisfiable	85	84	85
	unsatisfiable	60	63	67
	TOTAL	144	147	152
	50th (s)	195	172	131
	100th (s)	1193	1144	1060

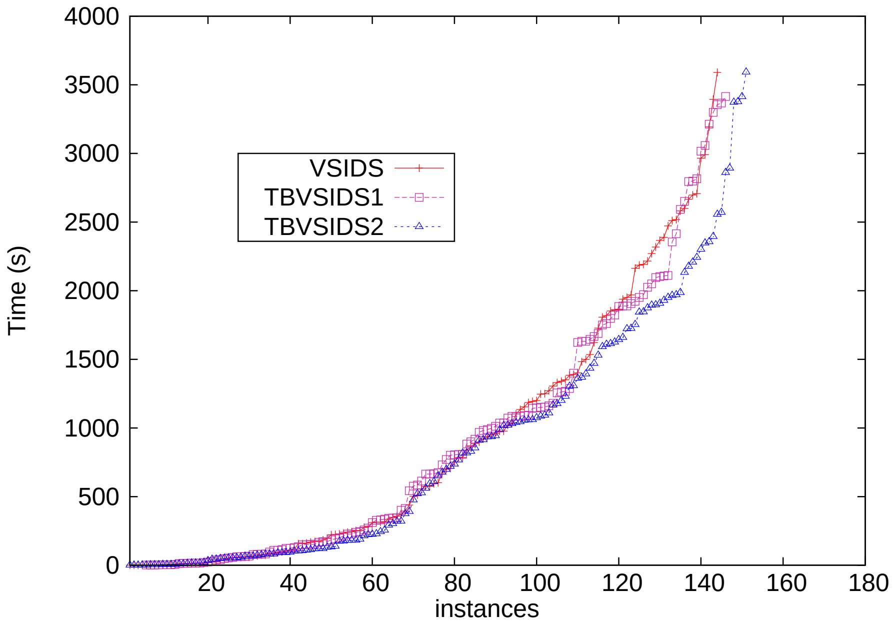


Fig. 4.3: Cactus plot of VSIDS and TBVSIDSs on MiniSat 2.2.

Cactus plot in Figure 4.3 compares performances among different branching heuristics. We cannot insist clearly that TBVSIDS1 is better than VSIDS in this figure, but the curve of TBVSIDS2 is definitely located at the right side of the other curves and showed better performance than other branching heuristics. The number of benchmarks used were 300, and solved instances by each branching heuristic were around 150. In this case, comparing median time would be inappropriate. Therefore, We compared 50th and 100th times by ascending order instread and showed their times in Table 4.2. Based on the comparisons of 50th and 100th data, TBVSIDS1 seem to be better than VSIDS. However, if we compare their curves in Figure 4.3, their performances seem to quite similar. However, if we consider the time for the additional calculation for breaking ties in TBVSIDS1, we might assume that TBVSIDS1 performed more intensive search than VSIDS.

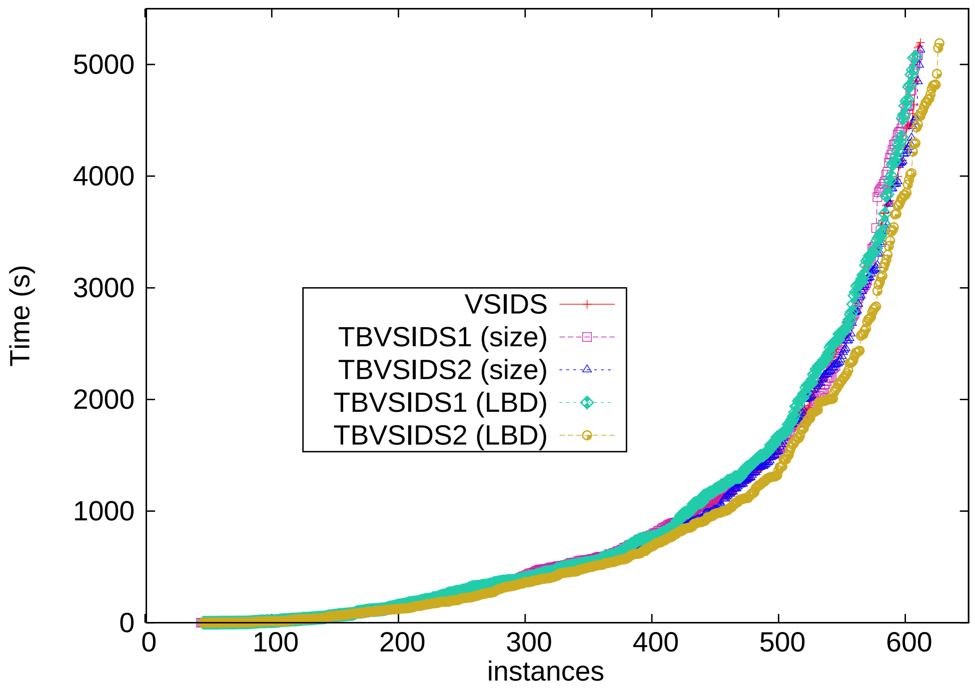
In the next step, we implemented TBVSIDSs in a modern solver Glucose 3.0. This solver is widely used as a base solver for many of modern solvers. We applied the same way as in Minisat 2.2 except for using a *LBD* indicator. A branching heuristic picks a variable that would have a high potential to induce a chain of unit propagations so that a new conflict occurs and learns a new clause. Therefore, we can consider that a branching heuristic is deeply connected to a clause learning scheme, and a varible appears a lot in short clauses might be preferred.

Table 4.3: Comparison of solved instances between VSIDS and TBVSIDSs from 900 instances of SAT Competitions.
(C: crafted track, A: application track, S: satisfiable instances, U: unsatisfiable instances, T: S + U)

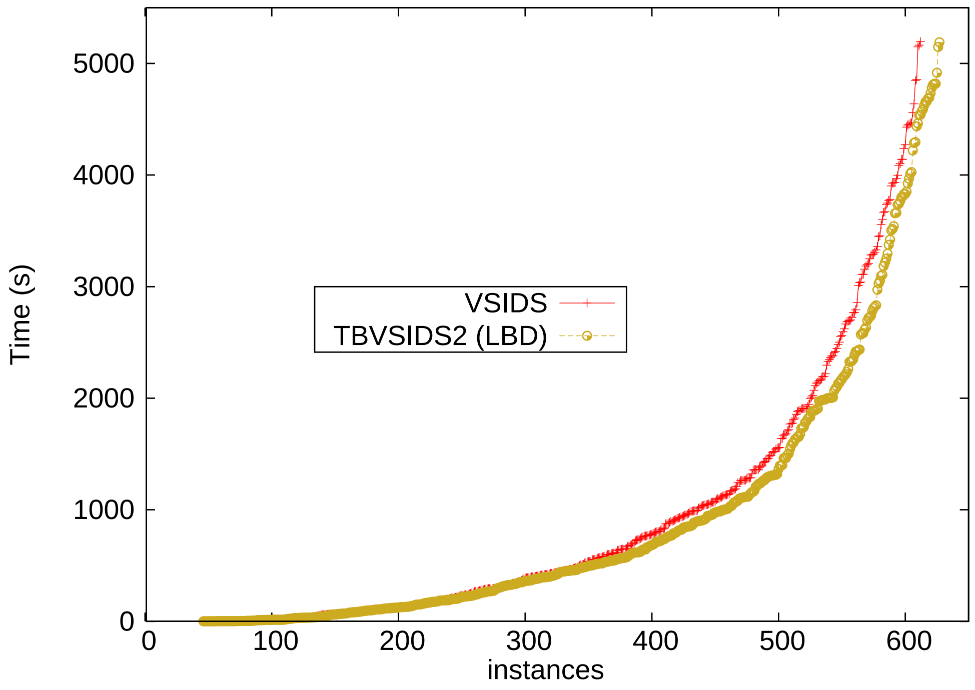
		Unmodified	TBVSIDS1 (*LBD*)	TBVSIDS1 (length)	TBVSIDS2 (*LBD*)	TBVSIDS2 (length)
2014C	S	79	82	81	81	80
	U	82	84	85	88	86
	T	161	166	166	169	166
2014A	S	100	100	98	102	101
	U	115	108	110	112	108
	T	215	208	208	214	209
2015A	S	137	140	139	143	140
	U	101	98	100	102	99
	T	238	238	239	245	239
TOTAL	S	316	322	318	326	321
	U	298	290	295	302	293
	T	614	612	613	628	614

As a solver learns new clauses during search, the most important variable changes dynamically. Therefore, we need to consider interplay between a branching heuristic and a clause learning scheme. Minisat 2.2 assesses learned clauses through their sizes. However, Glucose 3.0 uses *LBD* to assess learned clauses. Therefore, breaking ties using *LBD* indexes might be the better idea than using their sizes in Glucose 3.0. If we succeed to pick more influential variables in a branching heuristic, a solver performs more intensive search and the efficient reuse of learned clauses increase the number of unit propagations.

We evaluated TBVSIDSs using three different tracks, total of 900 instances, from SAT Competitions which are crafted track in SAT Competition 2014 and application track in SAT Competition 2014 and SAT-Race 2015. Time limit was set 5,000 s. Benchmark results are shown in Table 4.3 and Figure 4.4. If we only consider the number of solved instances among different branching heuristics using Table 4.3, only TBVSIDS2 with *LBD* indicator showed better results than those of VSIDS. We can observe the same results on their running times. Each curve in Figure 4.4 is sorted through running times of instances. In Figure 4.4.(a), we displayed results of all branching heuristics, but it was difficult to compare them because their performances are quite similar except for those of TBVSIDS with *LBD*. Therefore, we compared only VSIDS and TBVSIDS2 with *LBD* in Figure 4.4. TBVSIDS2 showed better performances than VSIDS no matter how we set up time limit,



(a)



(b)

Fig. 4.4: Cactus plot of VSIDS and TBVSIDSs on Glucose 3.0.

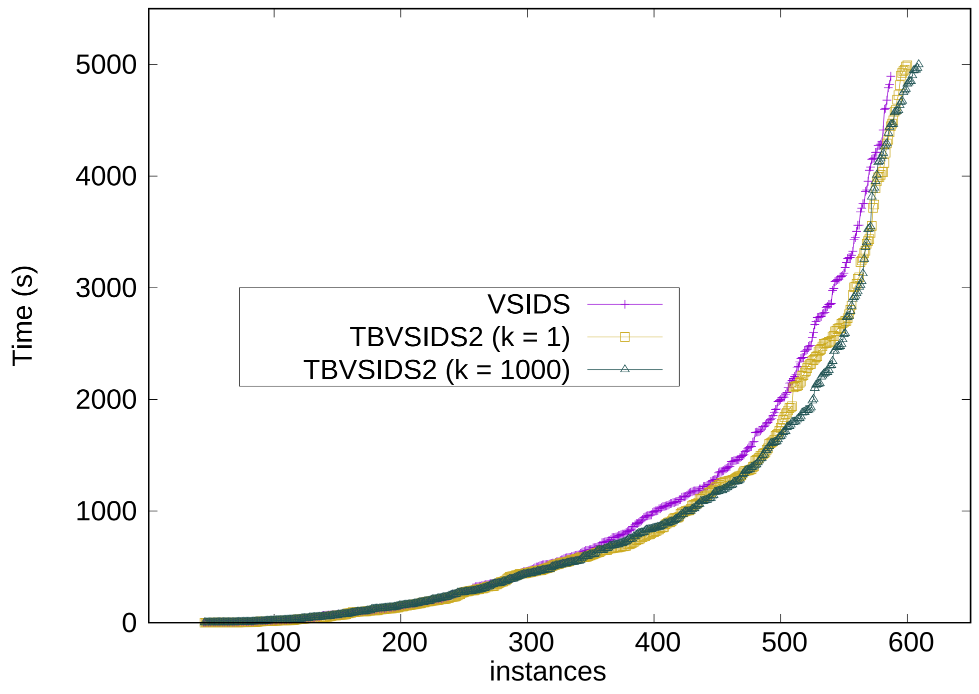


Fig. 4.5: Cactus plot of VSIDS and TBVSIDS2s on Glucose 3.0.

because the TBVSIDS2 curve is always located at the right side of the VSIDS curve.

All the TBVSIDS2 results described above were performed with $k = 1$. We above-mentioned adjusting k might be important in TBVSIDS2. Therefore, we adjusted the k value and tested using 900 instances stated above. Results are described in Figure 4.5 and Table 4.4. Generally, evaluating SAT solvers requires a lot of time, because hundreds of instances should be performed with the long time limitation. To reduce the burden, we tested them on 15 parallels this time. Therefore, results were little bit worsen than those in Figure 4.3. Solved instances in VSIDS and TBVSIDS2 default were reduced 24 and 26, respectively. By adjusting $k = 1000$, we were able to solve more instances. Differences between $k = 1$ and $k = 1000$ is slight, but entirely results of $k = 1000$ were better than those of $k = 1$.

Finally, we compare branching heuristics based on the original VSIDS. We remark EVSIDS as VSIDS throughout the paper except for this part. EVSIDS basically add scores to variables related to resolutions, i.e., conflict analysis, and we broke ties by giving bonuses to variables in a learned clause. However, the original VSIDS only add scores to variables in a learned clause, and they are incremented by 1. We implemented VSIDS$_S$

Table 4.4: Solved instances of VSIDS and TBVSIDS2s from 900 instances of SAT Competitions.
(C: crafted track, A: application track, S: satisfiable instances, U: unsatisfiable instances, T: S + U)

		VSIDS	TBVSIDS2 ($k = 1$)	TBVSIDS2 ($k = 1000$)
2014C	S	77	79	82
	U	78	84	86
	T	155	163	168
2014A	S	98	100	98
	U	107	103	109
	T	205	203	207
2015A	S	132	138	137
	U	98	98	99
	T	230	236	236
TOTAL	S	307	317	317
	U	283	285	294
	T	590	602	611

and VSIDS$_L$ by incrementing variables in a learned clause with the size and *LBD* of a clause, respectively. And we compared VSIDS$_S$ and VSIDS$_L$ with the original VSIDS to ensure the importance of interplays between a branching heuristic and a clause learning scheme. We also want to insist applying *LBD* to branching heuristics is an efficient method, because the qualties of learned clauses are evaluated by a *LBD* indicator in the state-of-the-art SAT solvers.

Table 4.5 and Figure 4.6 compares three branching heuristics abovementioned. Results clearly show that considering the qualities of learned clauses performs better than pure VSIDS. The performance gaps between VSIDS and the others are quite clear. We also displayed the performance of EVSIDS, and the EVSIDS definitely outperforms the original VSIDS and its variants. Because of its efficiency, EVSIDS is adopted in many modern SAT solvers and called VSIDS instead of the original VSIDS, and we have succeeded to improve EVSIDS by breaking ties with the consideration of interplay between a branching heuristic and a clause learning scheme.

4.3.3 Comparison between CHB and TBCHB

Results of CHB and TBCHBs are described in Tables 4.6 and 4.7. To reduce the burden, we tested them on 15 parallels in this section.

Table 4.5: Comparison of solved instances between the original VSIDS and its variants from 900 instances of SAT Competitions.
(C: crafted track, A: application track, S: satisfiable instances, U: unsatisfiable instances, T: S + U)

		VSIDS	VSIDS$_S$	VSIDS$_L$	EVSIDS
2014C	S	69	69	69	79
	U	56	65	64	82
	T	125	134	133	161
2014A	S	83	81	81	100
	U	86	93	96	115
	T	169	173	178	215
2015A	S	112	111	114	137
	U	84	87	91	101
	T	196	198	205	238
TOTAL	S	264	261	264	316
	U	226	242	252	298
	T	490	505	516	614

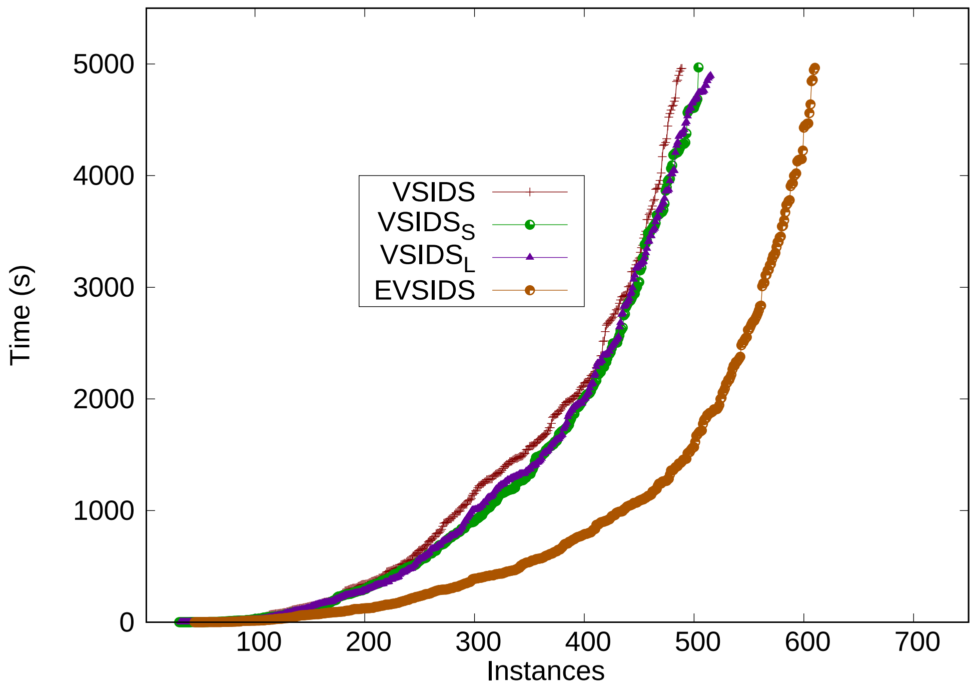


Fig. 4.6: Cactus plot of original VSIDS and its variants on Glucose 3.0.

First, we compared CHB and TBCHB with $k = 1000$. As $k = 1000$ showed the best results in the TBVSIDS experiments, we adopted this value for the comparison between CHB and TBCHB. Table 4.6 shows that results. VM is used for tests. TBCHB with $k = 1000$ performed better than CHB in our experiments. However, we might also try other ks instead.

Table 4.6: Solved instances of CHB and TBCHB from 900 instances of SAT Competitions. (C: crafted track, A: application track, S: satisfiable instances, U: unsatisfiable instances, T: S + U)

		CHB	TBCHB2 ($k = 1000$)
2014C	S	76	78
	U	72	81
	T	148	159
2014A	S	88	88
	U	87	89
	T	175	177
2015A	S	138	136
	U	86	90
	T	224	226
TOTAL	S	302	302
	U	245	260
	T	547	562

Table 4.7: Comparison of solved instances among TBCHBs from 900 instances of SAT Competitions.
(C: crafted track, A: application track, S: satisfiable instances, U: unsatisfiable instances, T: S + U)

		TBCHB2 ($k = 1$)	TBCHB2 ($k = 2$)	TBCHB2 ($k = 5$)	TBCHB2 ($k = 10$)	TBCHB2 ($k = 100$)	TBCHB2 ($k = 1000$)
2014C	S	81	73	78	80	83	80
	U	69	74	87	94	100	96
	T	150	147	165	174	183	176
2014A	S	93	99	100	99	102	102
	U	62	67	70	77	102	100
	T	155	166	170	176	204	202
2015A	S	138	137	138	143	147	147
	U	72	76	75	82	99	96
	T	210	213	213	225	246	243
TOTAL	S	312	309	313	322	332	329
	U	203	217	232	253	301	292
	T	515	526	545	575	633	621

We tested with several ks, and results are described in Table 4.7. In this time, we used WS for experiments, thus Table 4.6 and Table 4.7 is divided into two tables. Entirely, results get worse when k becomes small under 100. However, $k = 100$ showed better results than those of $k = 1000$. In this time, we did not compared CHB and TBCHB with $k = 100$ directly, their performance gap would be larger than a gap between CHB and TBCHB with $k = 1000$ based on our experiments. Further experiments at $10 < k < 1000$

might induce better results.

4.3.4 Analysis among branching heuristics

In Section 4.3.2, TBVSIDS2 with *LBD* index showed the better results than VSIDS. We compare several indicators during search between VSIDS and TBVSIDS2 with *LBD* index in this section.

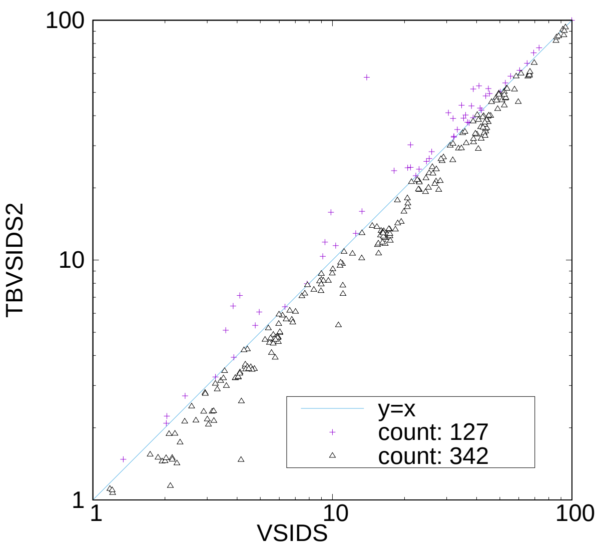
Figure 4.7.(a) compares tie occurrences between VSIDS and TBVSIDS2 through a scatter plot. Results are obtained using 900 benchmarks from SAT Competitions abovementioned in Table 4.3. Time limitation was set 1,000 s for each instance. Each instance was performed 3 times using VSIDS and TBVSIDS2, respectively.

We only compared instances those were unsolved within 1,000 s, thus only 469 instances were compared. We succeeded to reduce tie occurrences in 342 instances from 469 instances (72.9%). We were unable to reduce ties from all instances, because subspace changes after *tie-breaking* and this will lead different search. Therefore, we cannot assure ties would be reduced in all instances. It would be reasonable to compare the entire tendencies of ties using hundreds of benchmarks.

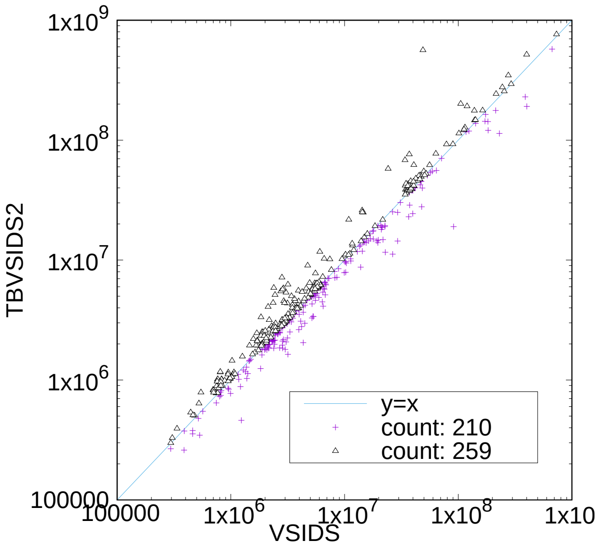
Reduced ties does not indicate better search. Therefore, we measured unit propagation rate for each instance from 469 instances in Figure 4.7. High propagation rate could be a good indicator to measure search efficiency of SAT solver, because this represents agility or productivity of a SAT solver. In Figure 4.7.(b), propagation counts for each instance between VSIDS and TBVSIDS2 are compared. We can observe propagation rates of TBVSIDS2 are higher that those of VSIDS entirely. Therefore, we can presume *tie-breaking* methods succeeded to pick more influential variable from ties and performed intensive search to induce frequent unit propagations.

To have confidence of our hypothesis, we measured distribution of learned clauses through their sizes and *LBD*s and provided in Figure 4.8 and 4.9. Conditions are the same we mentioned above, and distributions of clauses are measured for each instance and summarized. Instances solved within time limit by either of VSIDS or TBVSIDS2 are excluded.

Therefore, each pair of green and red boxes positioned side by side compares the number of learned clauses with size or *LBD* equals i, where $2 \leq i \leq 30$. We only displayed clauses



(a) Tie occurrences

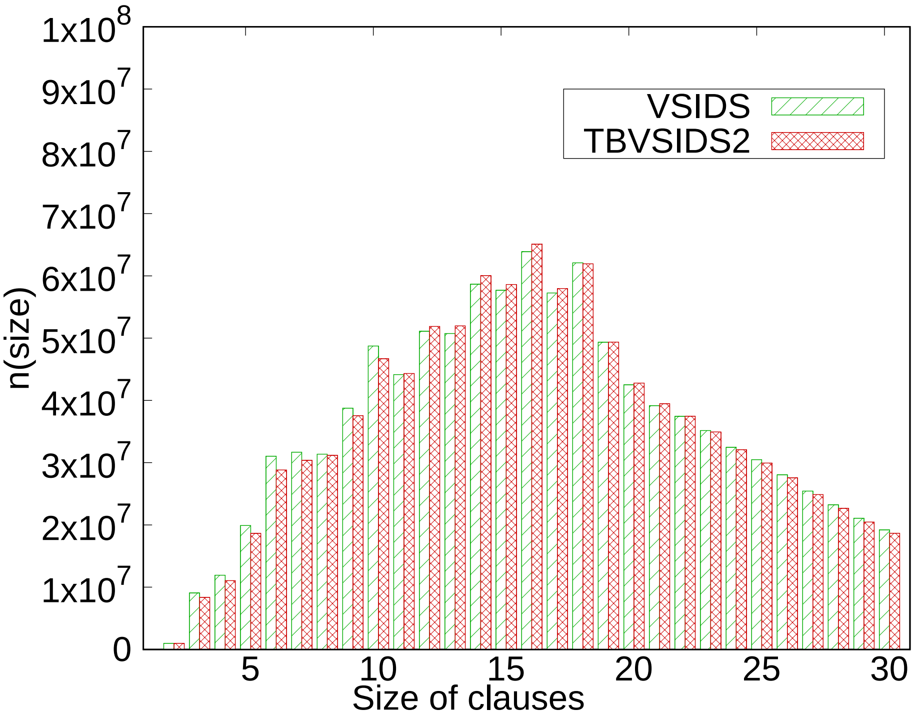


(b) Unit propagations

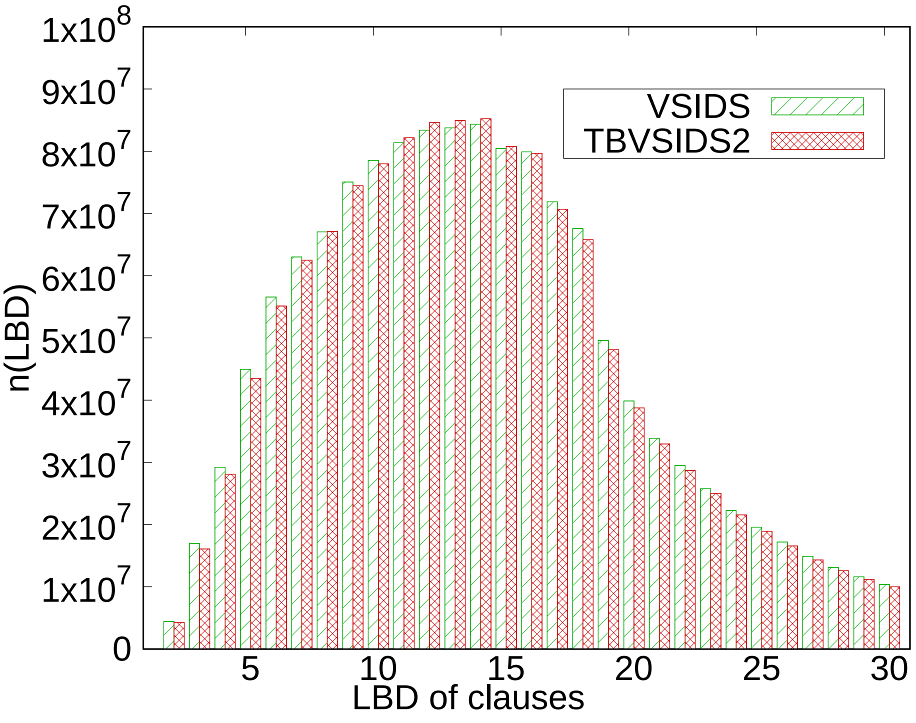
Fig. 4.7: Scatter plots for Comparisons of VSIDS and TBVSIDS2.

under 30 of size or *LBD*, because too long clauses have little possibility to become unit clauses. In general, short clauses such as binary, ternary clauses and clauses of $LBD =$ 2 are more informative than long clauses. Many short clauses are preserved, because they have high potentials to induce unit propagations or conflicts. We considered simply comparing the number of clauses with different sizes or *LBD*s would be unfair for short clauses. Therefore, we normalized their numbers by multiplying 2^{2-x}, where x is size or *LBD*.

Now we can clearly observe their distribution tendencies. If we look in Figure 4.9, TBVSIDS2 found more learned clauses than VSIDS when their sizes or *LBD*s are small. These results support our hypothesis. TBVSIDS2 demonstrated the better propagation rates than VSIDS, even though VSIDS found more number of short clauses. Entirely, the lengthes of learned clauses in VSIDS were shorter than those in TBVSIDS2. In general, short clauses are considered important because of their high potentials for unit propagations, and VSIDS found more of short clauses, but unit propagations were reduced in VSIDS. Therefore, we might conclude that TBVSIDS2 found less clauses but it succeeded to reuse learned clauses because of its intensive search.

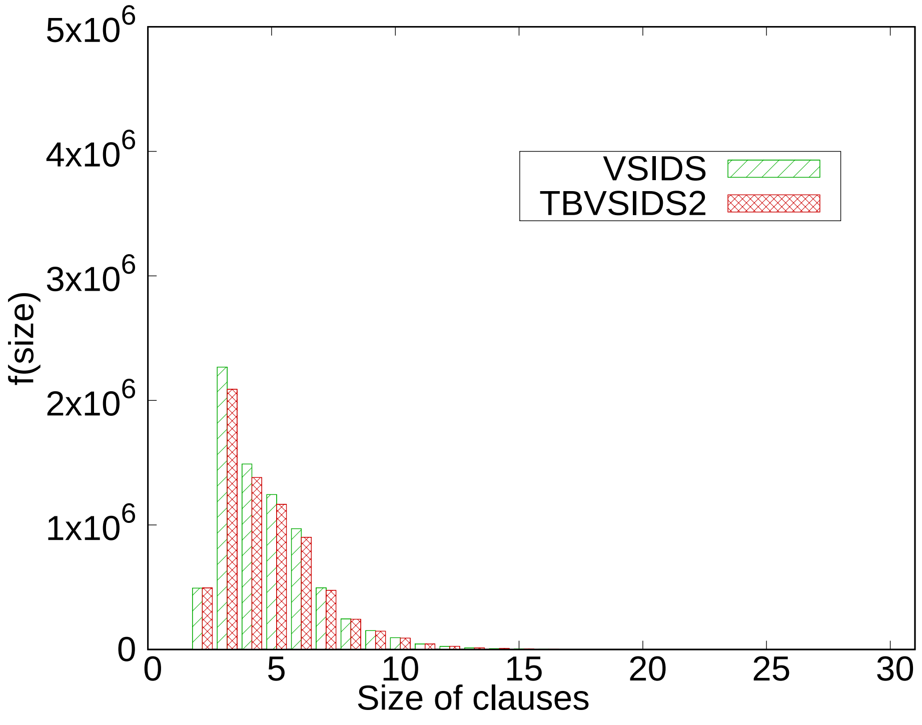


(a) size

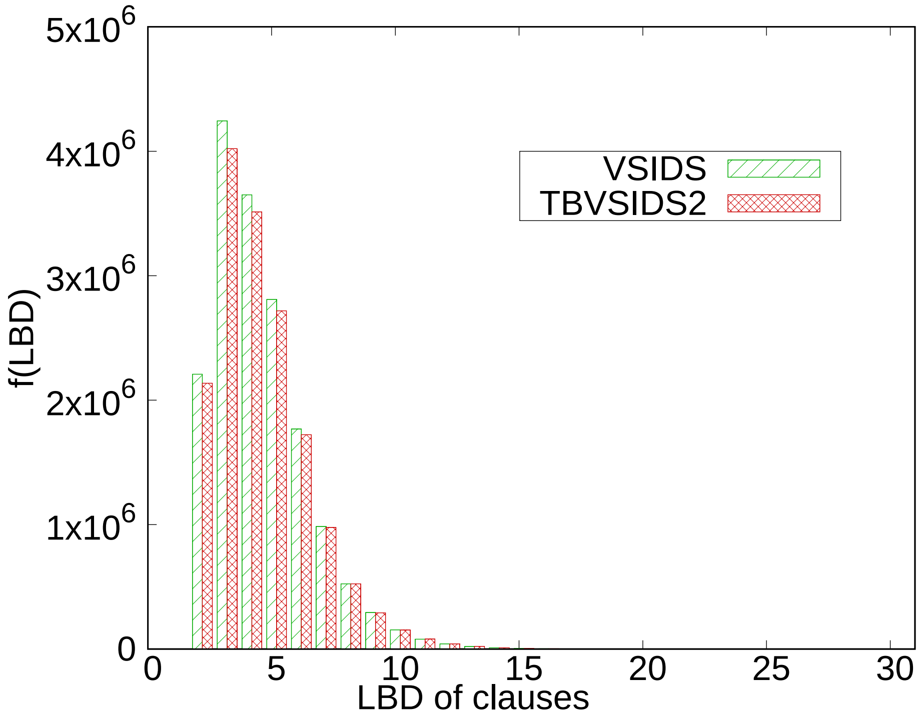


(b) *LBD*

Fig. 4.8: Comparison of size and *LBD* distribution between VSIDS and TBVSIDS2 n(x) is the number of learned clauses found during search.



(a) size (normalyzed)



(b) *LBD* (normalyzed)

Fig. 4.9: Comparison of size and *LBD* distribution between VSIDS and TBVSIDS2 normalyzed by $f(x) = n(x) \times 2^{(2-x)}$, where x is size and *LBD* respectively.

Chapter 5

Hybrid branching heuristic

In the previous chapter, we proposed to apply *tie-breaking* for branching heuristics. We are considering *tie-breaking* might become a universal method for improving branching heuristics. However, there is a certain limit to the range of instances a single branching heuristic can solve. In this chapter, we propose a hybrid branching heuristic, as an effort to overcome the limitations of single branching heuristics. This chapter describes our first step of algorithm selection research for integrating the performances of SAT solvers. Our midterm objective is to provide a widely applicable model of hybrid branching heuristic instead of VSIDS. Our final goal is to integrate several hybrid models in a single-solver for ongoing research.

This chapter consists of as follows. First, we provide previous works of algorithm selection for SAT solvers, address an issue of multi-solver approaches, describe our approach for integrating the performance of SAT solvers, and emphasize the necessity of our proposal. Second, we create a static model of hybrid branching heuristic using two branching heuristics, and compare the performances among a hybrid branching heuristic and single branching heuristics as a preliminary experiment. Third, we create a hybrid model using eight branching heuristics. To achieve this, we adopted to use random forest, extracted 13 features from SAT formulas, and gathered experimental results for each branching heuristic. Fourth, we propose a random sampling to reduce the time of extractions of several features which are infeasible to extract for several large instances. Finally, we evaluate a model by expanding the number of features to 23 and applying the random sampling, and propose for applying a genetic algorithm to find an efficient model within a reasonable timeframe.

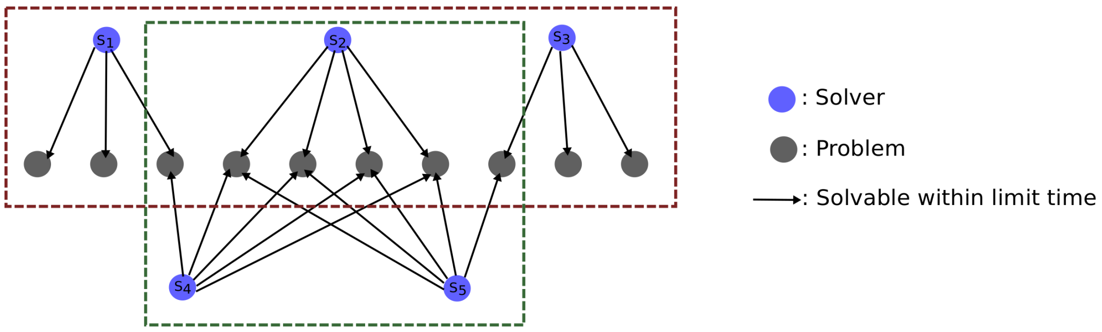


Fig. 5.1: Set covering problem between solvers and instances

5.1 Backgrounds - Algorithm selection in SAT research

The concept of algorithm selection problem [101] is proposed with the intention of constructing a model to obtain the high performance. There are several SAT related algorithm selection studies, because SAT solvers handle NP-complete problems; thus, there are no polynomial-time algorithms. Therefore, a single algorithm works well with some instances and works bad in other instances.

We can adopt the "winner-take-all" approach, i.e., select an algorithm having the best performance. However, if we are able to select an appropriate algorithm for each instance, this approach would be an ideal solution. We give an example in Figure 5.1 by considering the algorithm selection problem as a set covering problem. If we pick three solvers greedily based on their numbers of solved problems, solvers S_2, S_4, and S_5 would be picked. However, they can cover only 6 in 10 problems because they solve similar problems. If we pick three solvers S_1, S_2, and S_3 instead, we can cover all given problems. Therefore, different solvers must have different performances and adequate features are required to differentiate them.

The most representative solver is a multi-solver SATzilla [16]. It creates a ridge regression model to predict a logarithmic runtime of a SAT instance for each solver. It provides satisfiable and unsatisfiable models respectively and predicts the probability of being satisfiable for a given instance to select an adequate model. As several features require a long time for extraction, there exists a 60 sec limit for extracting features per each instance. If a solver reaches time limitation for feature extraction, it runs a backup solver.

A backup solver is prepared greedily by "winner-take-all" approach from benchmark results. SATzilla also prepare pre-solvers before feature extraction. Pre-solvers are run very shortly (under 5 seconds) to maintain the agility of a solver. It selected 7 different solvers manually from existing solvers. SATZilla demonstrated its efficiency in SAT Competition 2007.

Kadioglu *et al.* [102] proposed to utilize k-nearest neighbor classification for algorithm selection. They extracted 48 features, which are the same in SATZilla, and calculated the Euclidean distance between instances using [0, 1] normalized feature values. The value of k is obtained by selecting the best performing k from iterations of random sub-sampling validation. A solver with the best PAR10 score in k nearest instances is assigned when an instance is given to its model. The k-NN model showed slightly better performance than SATZilla, even though it does not require to learn any sophisticated model such as a runtime prediction. Kadioglu *et al.* also proposed to bulid a solver schedule that indicates a sequence of solvers with their time limits respectively to solve a problem. They proposed this to increase the efficiency of a multi-solver, because an instance can be solved in a short time by one solver when it cannot be solved by other solvers for a long time. Clustering method using the Euclidean distances have some weaknesses. If variances of unimportant/important features are big/small, they will deteriorate the efficiency of clustering.

SATZilla 2011 [103] improved algorithm selection core while maintaining rest parts such as a backup solver and pre-solvers. They applied pairwise cost-sensitive decision forests instead of ridge regression. A Decision forest $DF(i, j)$ were trained for each pair of solvers (S_i, S_j), and it votes for either of them. Cutoff time of feature extraction was set to 500 s for each instance.

Cost-sensitive hierarchical clustering (CSHC) [104] applied a single random forest to costruct a model for a multi-sover. They compared performance of a model by changing bootstrap aggregating policies, the number of clustering, adding combination features, and merging of clusters. Their experiments of changing policies in a random forest provided not a significant results, because there was no clear winner among policies or if there was a slightly improved performance, the results were unstable. However, CSHC demonstrated its efficiency, even though its training time is much faster than SATZilla 2011.

A deep learning approach has also been attempted [17]. This approach converts a CNF into a grayscale image and builds a classifier using a convolutional neural network. The input layer is a converted grayscale image with a standard image scaling operator, and the output layer provides probabilities of N solvers. They insisted extracting good features is arduous work, and a CNN approach can be utilized genereally even for non-experts. However, authors remarked this approach might have the robustness issue such as clause re-ordering.

Researches stated above required several state-of-the-art solvers. They achieved higher performances than single solvers, but they are unsuitable for acting as a base solver, because they already include several base solvers.

Michail *et al.* [105] proposed a reinforcement learning approach to select one from several branching heuristics based on each instance for #SAT. The #SAT solver counts the number of satisfying assignments for a given SAT formula. They learned a value function $Q(s, a)$, where s and a are state and activity, respectively. A state is defined by the number of variables in a formula, and an activity is the selection of a branching heuristic. After learning, a solver switches among branching heuristics dynamically based on subspaces. Authors in this paper addressed several limitations. They limited the state-space only using a feature of the number of variables, because a state has to be calculated rapidly. They anticipated a good state description may refine branching rules, but extracting good features requires significant computational cost.

Oh [106] analyzed the differences between satisfiable and unsatisfiable instances. Roles of learned clauses, restart policies, and decay factor in VSIDS are scrutinized in their studies. It is well known factor that a slow Luby restart policy is superior to rapid restart policies for satisfiable problems. The author simply designed a hybrid restart strategy by switching two phases of Luby restart and glucose restart. We agree with this idea even though the author remarked this is a very crude strategy.

Our objective is to create an efficient single-solver for ongoing researches. This solver has several hybrid models and selects policies on an instance basis. We considered a solver can improve its performance by combining several robust models. Each model focuses on only a small part. Figure 5.2 illustrates our idea to build an efficient SAT solver. Consider there are N and M numbers of different branching heuristics and restart

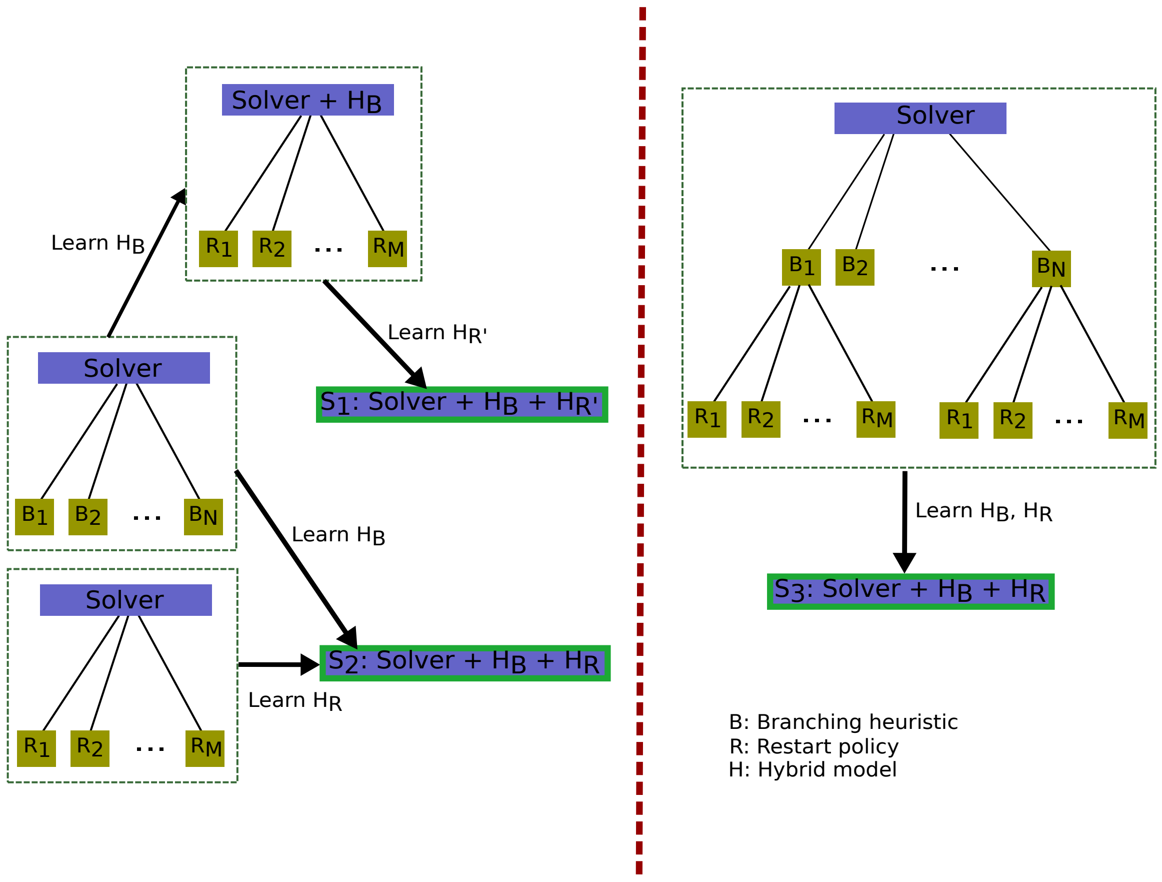


Fig. 5.2: Approaches for integrating performance of SAT solvers

policies, respectively. We propose to find a model by focusing on only a small part such as branching heuristics or restart policies. We can obtain final solvers S_1, S_2, and S_3 in Figure 5.2. We can learn both S_1 and S_2 with $O(N+M)$ policy space. However, learning S_3 requires $O(NM)$ policy space, many experimental results are required, and we concluded this is infeasible approach. We only illustrated two sorts of algorithms, but the size of policy space would be too large to learn for S_3 if we also consider other sorts of algorithms. There might be an efficiency problem in S_2, because it simply integrates different models. However, if we might create more general models by evaluating each model with several different solvers.

Here we remark our contributions in this Chapter. First, we propose a concept that can be implemented in a single-solver. This is not a multi-solver but a single-solver with hybrid models. Therefore, a solver with our proposal has a potential to be a base solver for continuous improvements. Second, we propose to apply a random sampling for feature extraction of SAT formulas. Several features such as those in a variable graph require a long time to extract and infeasible for large instances. We apply random sampling and demonstrate its validity empirically. Third, we propose a genetic algorithm to obtain an efficient model within a reasonable timeframe.

In the following sections, we first apply a static method using two branching heuristics as a preliminary step to observe the performance of a hybrid branching heuristic. Next, we briefly explain details of our model and a feature list. We construct a random forest model using 13 features, which can be obtained in a relatively short time. Next, we propose a random sampling to reduce the time for feature extraction. Finally, We expand a feature space to 23 features and apply a genetic algorithm and a random sampling to lead better result than those in 13 features.

5.2 Preliminary design - Static method with two branching heuristics

In this section, we build a hybrid model using two heuristics, i.e., TBVSIDS and CHB. Experiments in this section was aimed at observing the validity of a hybrid branching heuristic. We chose TBVSIDS and CHB, because their algorithms are basically different, thus integration of their performances might be much better than single of them. As they also use the same data structure, it is easy to implement them in a solver.

Our hybrid branching heuristic works as a preprocessing method using a pre-trained model for selecting a branching heuristic based on each SAT instance, because applying a dynamic solution that switches among branching heurisitics is too high-demensional to optimize. We manually compared performances of TBVSIDS and CHB using several features of SAT instances and applied a simple model to enhance the performance of a hybrid branching heuristic.

5.2.1 Difference between TBVSIDS and CHB

We compared the performance between TBVSIDS and CHB using several features of SAT formulas as an attempt to find a feature which is highly correlated with the performance gaps between TBVSIDS and CHB.

We utilized the number of variables, clauses, and communities and combined them by producing such as ratios, quotients, and products. We provide several results of them in Figure 5.3, 5.5, and 5.4.

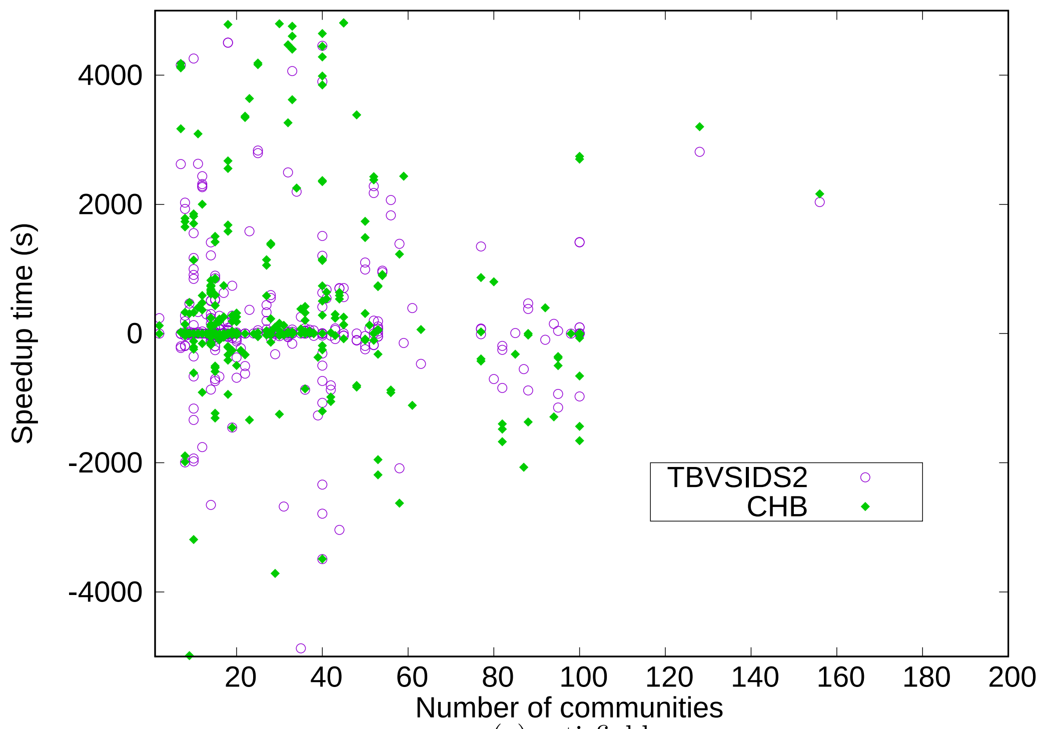
The speedups of CHB and TBVSIDS over VSIDS versus number of communities, ratio of clauses/variables, and number of variables in the SAT formula are shown in Figure 5.3, 5.5, and 5.4, respectively. Panels (a) and (b) of these figures plot the performances of satisfiable and unsatisfiable instances, respectively.

Instances were divided into satisfiable and unsatisfiable ones, because a variance of performance time is very large in the satisfiable instances. Therefore, we considered concentrating on observating the difference for only unsatisfiable instances might be better idea. This does not express we do not consider the performance for satisfiable instances. We can determine a policy using data of unsatisfiable instances, and if the performance for satisfiable instances with this policy is deteriorated, we can reject this policy.

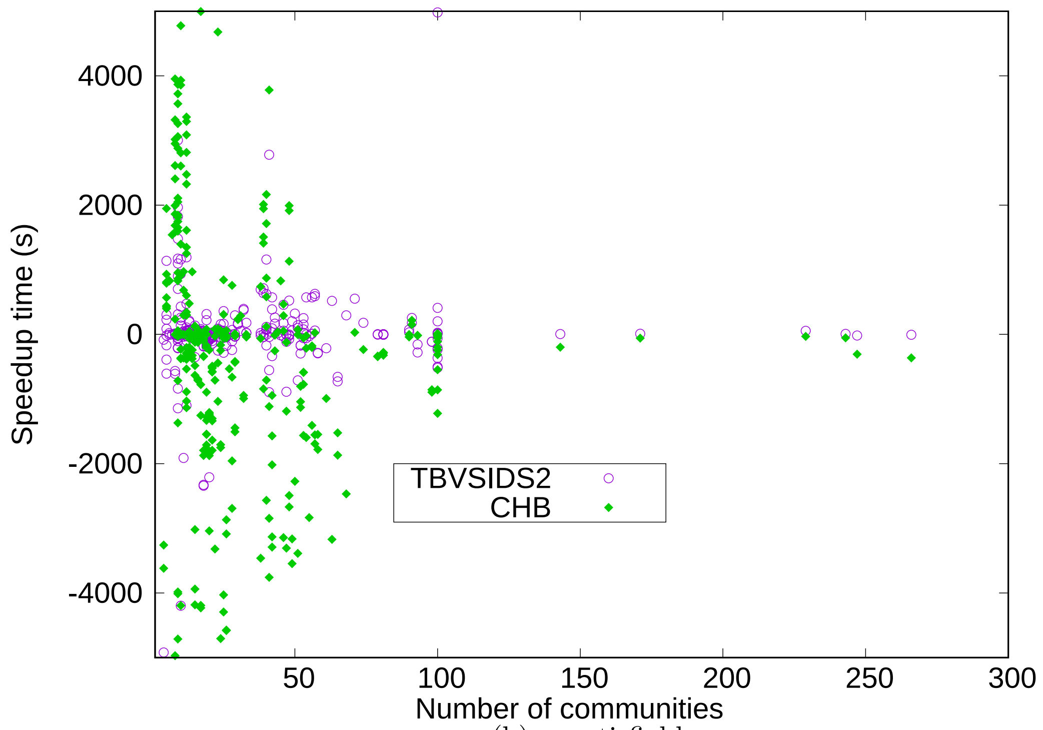
The results show that CHB and TBVSIDS2 are relatively distant from and close to VSIDS, respectively. CHB shows its differences over VSIDS clearly when we observe the results of unsatisfiable instances. It was difficult to observe the correlation between the number of communities and speedup time in Figure 5.3.(b), but CHB apparently performs well when the input formula has a small number of variables in Figure 5.4.(b). Determining a policy using satisfiable results looks difficult at least on figure we provided.

5.2.2 Experimental results

We implemented CHB in Glucose 3.0 and selected a branching heuristic as a preprocessing method. This method counts the number of variables, and selects CHB if the variables are fewer than 9,000 (switch line in Figure 5.4.(b)); otherwise, it selects TBVSIDS as its branching heuristic. The results are shown in Table 6.1 and Figure 5.6. Both TBVSIDS2 and CHB outperformed VSIDS, but the differences were not large. The results were

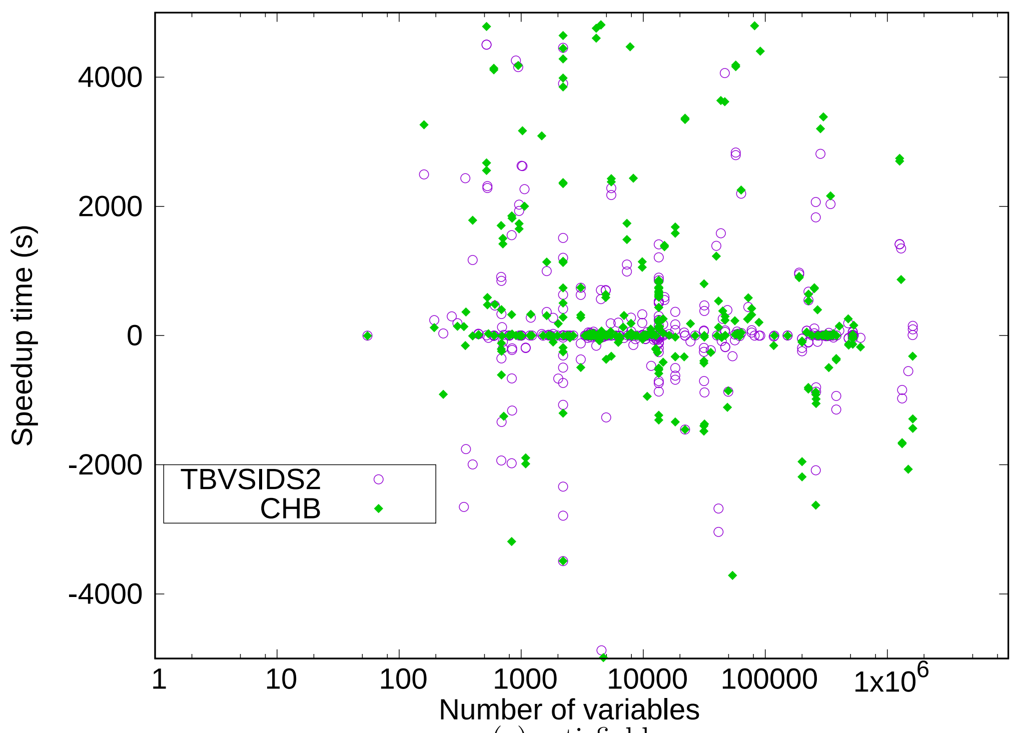


(a) satisfiable

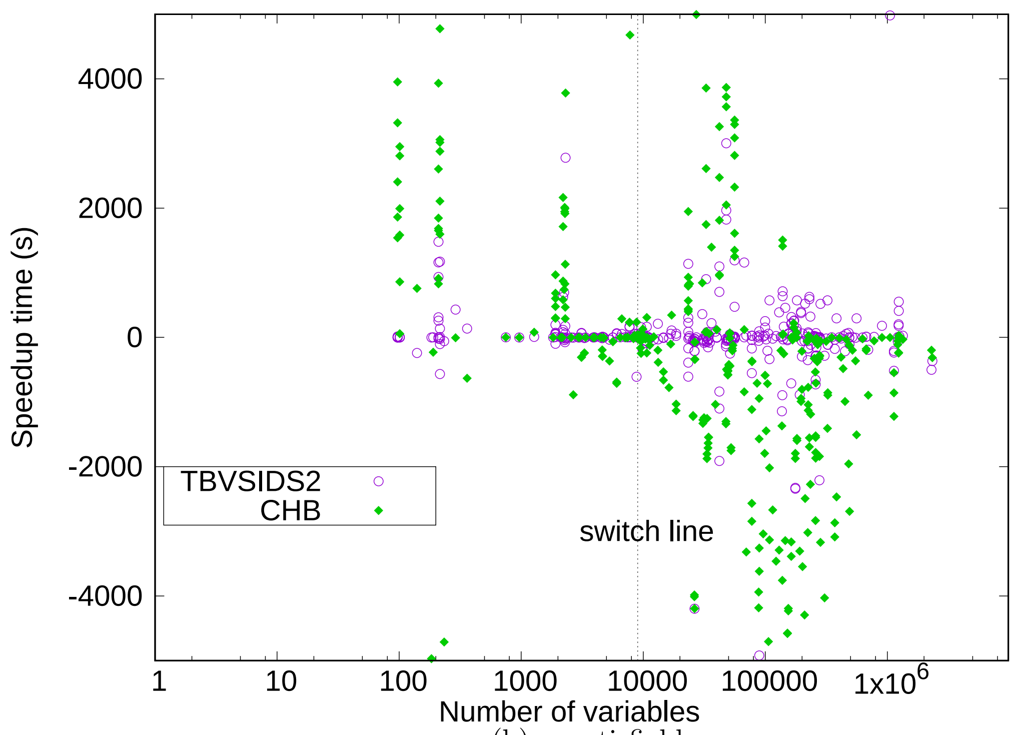


(b) unsatisfiable

Fig. 5.3: Speedup of CHB and TBVSIDS2 over VSIDS for the number of communities

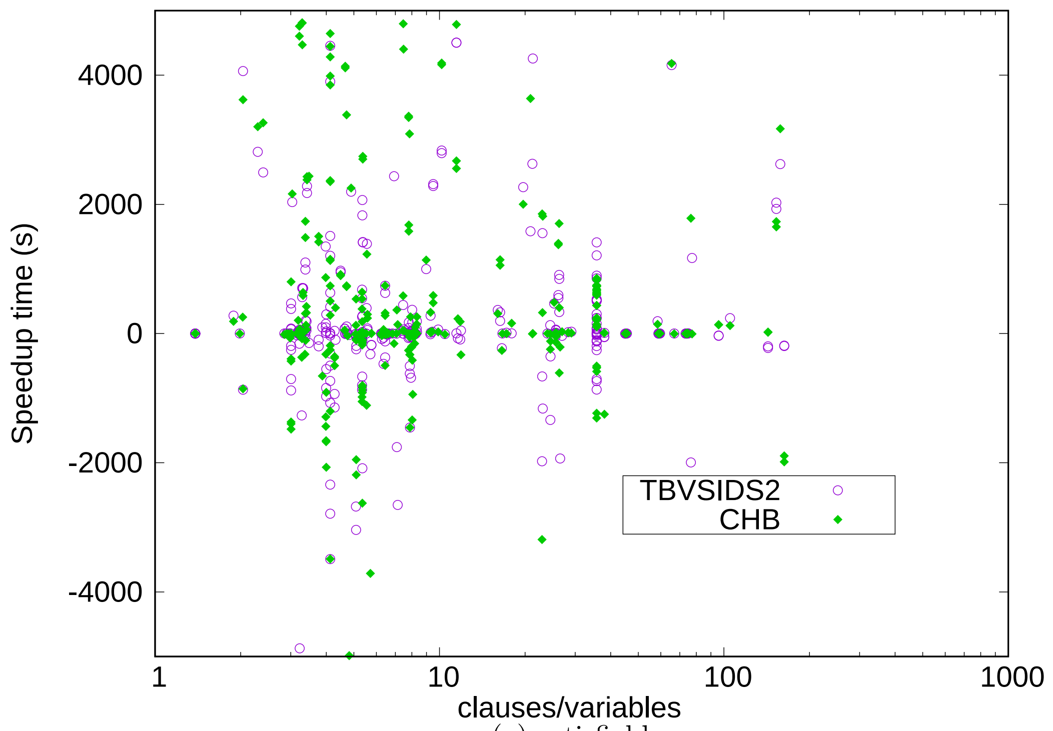


(a) satisfiable

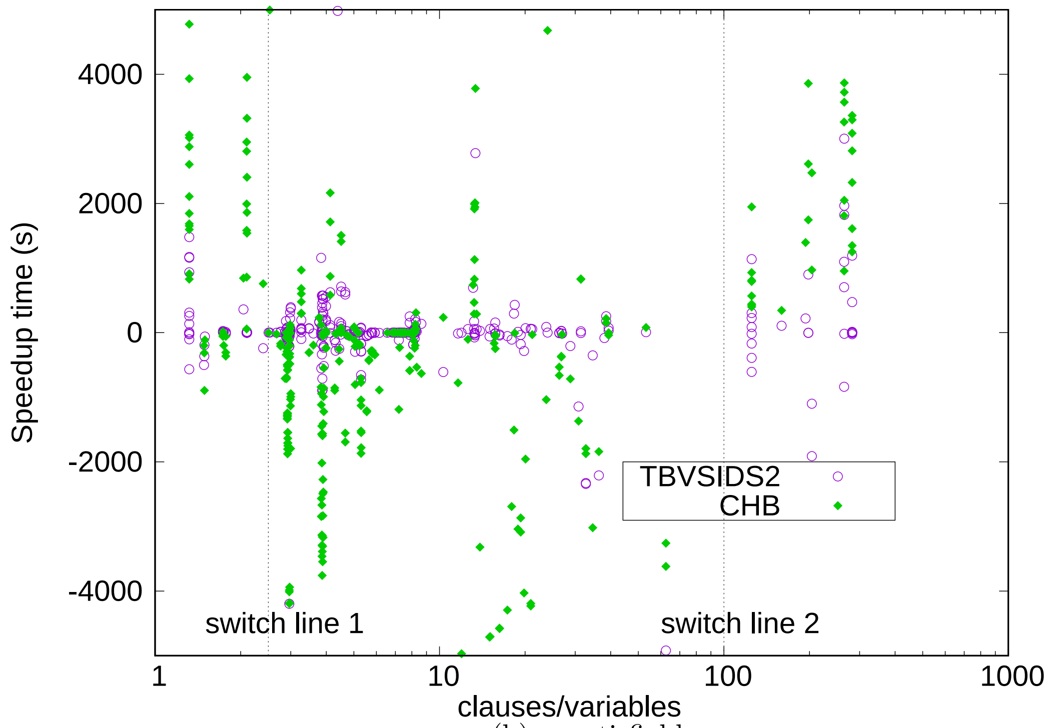


(b) unsatisfiable

Fig. 5.4: Speedup of CHB and TBVSIDS2 over VSIDS for the number of variables



(a) satisfiable



(b) unsatisfiable

Fig. 5.5: Speedup of CHB and TBVSIDS2 over VSIDS for the ratio of clauses/variables

Table 5.1: Solved instances from 900 instances of SAT Competitions. Column U, T2, C, and V denote unmodified original VSIDS, TBVSIDS2, CHB, and VBS, respectively. Rows C and A denote Crafted Track and Application Track, respectively.

Solver		U	T2	C	T2+C	V
2014C	satisfiable	79	81	85	83	89
	unsatisfiable	82	88	99	111	114
	BOTH	161	169	184	194	203
2014A	satisfiable	100	102	102	103	108
	unsatisfiable	115	112	102	115	123
	BOTH	215	214	204	218	231
2015A	satisfiable	137	143	146	146	152
	unsatisfiable	101	102	96	103	107
	BOTH	238	245	242	249	259
TOTAL	satisfiable	316	326	333	332	349
	unsatisfiable	298	302	297	329	344
	BOTH	614	628	630	661	693

improved by applying a hybrid method with an extremely simple policy. In fact, the results of the hybrid method exceeded our expectations. TBVSIDS2 and CHB solved 14 and 16 more instances than VSIDS, respectively. Therefore, we considered the gap between VSIDS and hybrid heuristic would be 30 at most (the arithmetic sum of 14 and 16). The actual gap was 47, indicating that TBVSIDS2 and CHB are complementary algorithms.

VBS in Figure 5.6 was constructed by a combination of CHB and TBVSIDS2. There remains a performance gap between VBS and a hybrid branching heuristic. Actually, we could create the better static model by simply combining switch line in Figure 5.4.(b) and switch line 1 and 2 in Figure 5.5.(b). However, our objective here was to check the validity of a hybrid branching heuristic and succeeded to improve the performance of a SAT solver largely. We were able to improve the performance of a SAT solver by applying a simple policy using only two branching heuristics.

VBS is an ideal solver and an actual solver cannot perform such as VBS. However, we can improve VBS by additionally combining several branching heuristics with CHB and TBVSIDS2. To integrate several branching heuristics, more complicated model would be required. In the following section, we add several branching heuristics for the better VBS and propose a random forest model to achive the better performance.

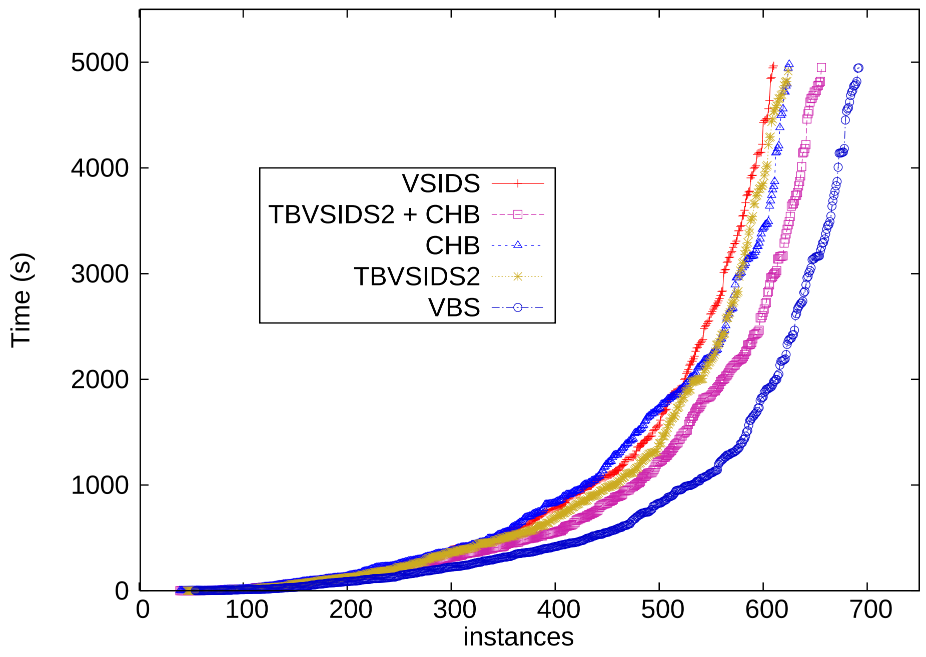


Fig. 5.6: Performance comparison of hybrid and single branching heuristics using 900 instances from SAT competitions

5.3 Random forest model - Eight branching heuristics with *tie-breaking*

In this section, we propose applying a random forest model to create a hybrid branching heuristic. We are motivated to build a hybrid model, because a single branching heuristic cannot cover all SAT instances. Let us consider the integration of N SAT solvers (S_1, S_2, ... , S_N) with different strategies, such as restart, learning scheme, and learned clause evaluation, into a solver I. We then optimize I. To improve I with a new policy P, we must implement P in each S_i and evaluate each case. In addition, after updating several S_is with P, we must rebuild a model for I. Applying and evaluating a new method seem to require much effort. Our final goal is to propose a base solver I with high performance and base-solver capability for other solvers such as MiniSat [58] or Glucose [44]. Such a base solver would allow continuous improvements of SAT solvers.

Existing algorithm selection strategies are only concentrated to improve their performances. They cannot be a base solver for other solvers. We considered to improve the performance of a solver by focusing on only a small part and improving that part within a reasonable timeframe. The improved solver can then be used as a base solver for other solvers. We considered that branching heuristics is an appropriate candidate for this purpose. Therefore, creating a hybrid branching heuristic is a first step to integrate the performances of modern SAT solvers such as S_1 shown in Figure 5.2.

Details of this section are as follows. First, we illustrate the list of branching heuristics, extracted features, and the description of random forest. Second, we construct a random forest model using 13 features that can be obtained within a reasonable timeframe. Third, an experiment of random sampling for feature extraction is considered as an attempt to reduce the extraction time for more complicated features. Finally, we expand features from 13 to 23 by applying random sampling and adopted the genetic algorithm to find an efficient model for two different SAT solvers.

5.3.1 Details for random forest

We first describe how a random forest model M works in a SAT solver. We prepare a set of branching heuristics B as classes for M and learn M through several features $F(\pi)$ in a SAT formula π and the running times $t(b_i)$ by branching heuristics $b_i \in B$ for training data. When a solver is requested to solve π', M in a solver classifies π' to assign a $b_{i'}$ for the branching heuristic based on $F(\pi')$. We do not prepare the default branching heuristic or cutoff time for feature extraction as an effort to provide general method by eliminating exceptions.

We arbitrarily determined details of random forest for constructing M because the objective in this section is not aiming for optimizing M. Indeed, optimizing M is not important in our concept. A model M should be required to combine with other M's such as a random forest model for restart policies.

The random forest is constituted by 30 decision trees. Training data Π is given to each decision tree dt_j. Each dt_j randomly selects and trains using half of Π. An index is provided to M to enable/disable each feature. Max depth of a tree is set to 5, and the minimum sample number for each node is two. At first, all training sample is in a node. If a node satisfies at least one of the termination criteria, it become a leaf node and partition process for this node is terminated. Terminiation is processed when samples in a node

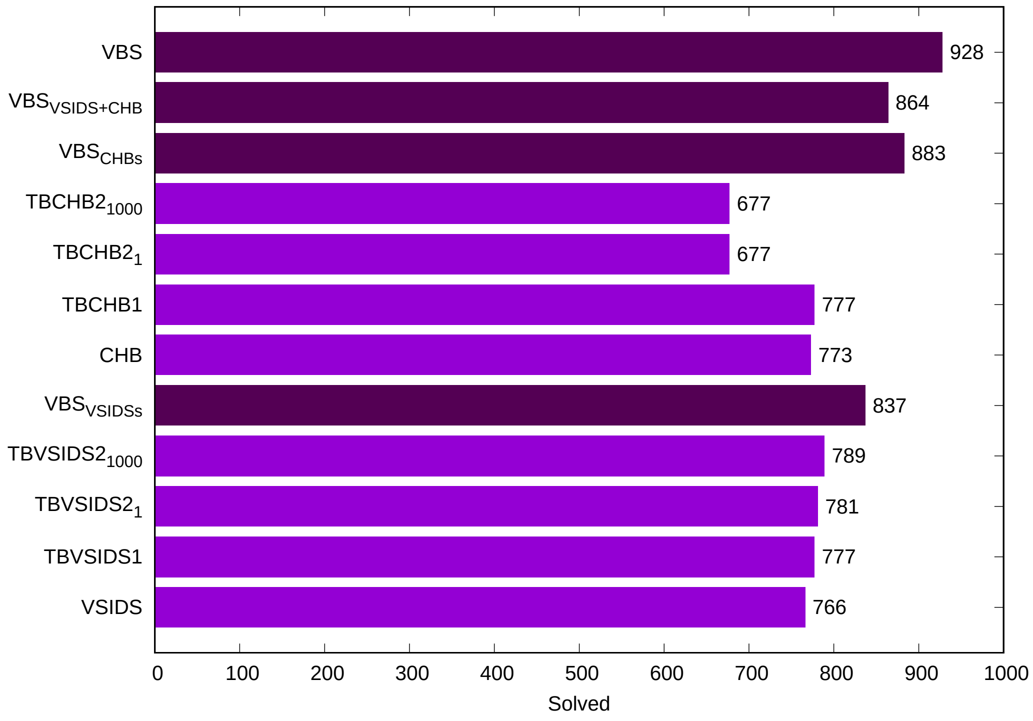


Fig. 5.7: Solved instances by each branching heuristic.

is below two, the height of a node reaches at five or impurity is zero. Misclassification impurity is adopted, and gain is obtained using simple distance criteria. Each node have a probability array for each class b_i.

Bootstrap aggregating is adopted with a winner aggregation. When π is given to M, a model iterates processes for each decision tree. For each dt_j, a b_i having the largest probability is chosen, and b_k that was chosen most is labeled to π.

Total of eight different branching heuristics were used as classes, which are VSIDS, CHB, TBVSIDSs, and TBCHBs. Figure 5.7 and 5.8 describe the performances and runtime correlation coefficient matrix for pairs of branching heuristics.

Eight branching heuristics were implemented on Glucose 3.0 and tested using total of 1400 benchmarks from SAT Competitions. Instances of both the Crafted and Application Tracks from 2014 to 2016 were used as benchmarks. Branching heuristics were generated based on VSIDS, CHB, and *tie-breaking*. We applied *tie-breaking* into VSIDS and CHB and generated TBVSIDSs and TBCHBs, respectively.

For TBVSIDS2 and TBCHB2, we assigned a different parameter for *quality* calculation, yielding two different TBVSIDS2s and TBCHB2s, respectively. The numbers 1 and 1000

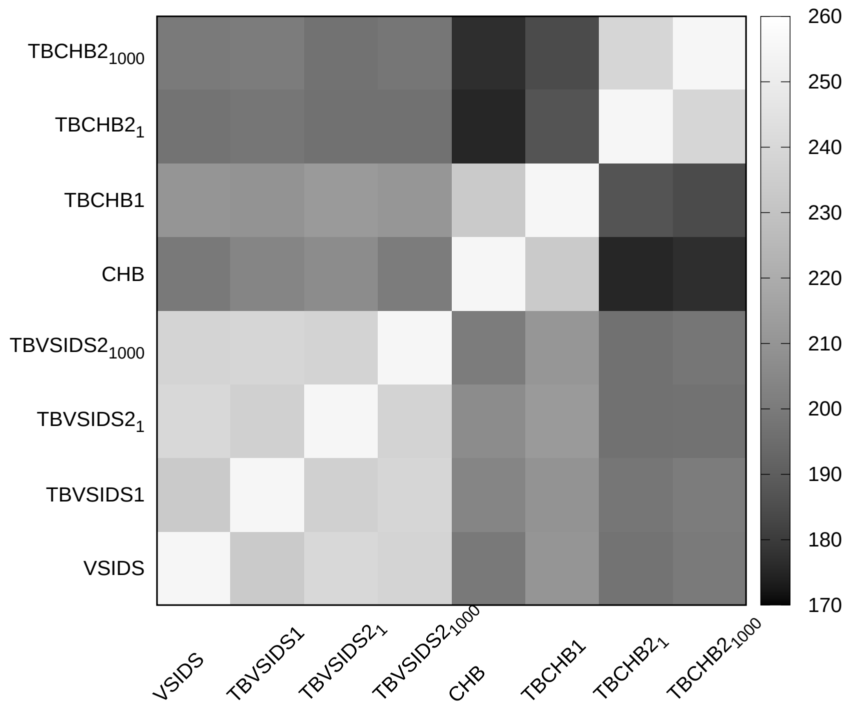


Fig. 5.8: Correlation matrix of runtimes among branching heuristics

in TBVSIDS2s and TBCHB2s indicate the value of k at line 11 in Algorithm 4. We improved TBCHB2 heuristic by adjusting the scale of *quality* in Algorithm 5, but experiments here were performed before the improvements of TBCHB2 heuristic. Therefore, $maxD$ in Algorithm 5 was fixed 1 here.

Figure 5.7 compares the number of solved instances from 1400 benchmarks. Dark and light purple boxes indicate several VBSs and results of eight branching heuristics, respectively. Figure 5.8 illustrates correlations of each pair of branching heuristics. Between VSIDS and TBVSIDS1 or CHB and TBCHB1 showed quite similar results, because TB-VSIDS1 and TBCHB1 maintain the original variable scores and break ties directly with another scores. In constrast to TBVSIDS2s, TBCHB2s deteriorated the performance of CHB because scale part for *quality* is not optimized here as we abovementioned. Therefore, correlations between (CHB, TBCHB1) and (TBCHB2$_1$, TBCHB2$_{1000}$) were relatively smaller than others. Consequently a hypothetical solver VBS$_{CHBs}$ was better than VBS$_{VSIDSs}$. As there exists a performance gap between VBS and VBS$_{VSIDS+CHB}$, it is meaningful to construct a model using these eight branching heuristics.

We compare VBS with top solvers of SAT Competitions in a Table 5.2 to observe the

Table 5.2: Comparison of performances between VBS and top solvers in SAT Competitions. Rows C and A denote Crafted Track and Application Track, respectively. Each solver in column 4 is a top solver at a SAT competition in column 1. CSS denotes a combination of single solver by selecting a best solver for each tracks.

	Solver	Solved	Best Solver	Solved
2014C	VBS	214	glueSplit_clasp	208
2014A		231	Lingeling ayu	231
2015A		261	abcdSAT	261
2016C		65	tc_glucose	58
2016A		157	MapleCOMSPS_LRB_DRUP	154
TOTAL		928	CSS	912

potential of a hybrid branching heuristic. Results of VBS were generated from 8 branching heuristics abovementioned and obtained by testing them on WS with time limit 5,000 s. Results of best solvers were brought from results of SAT Competitions. Each solver in the column of Best Solver is a winner in each competition. For example, the glueSplit_clasp and Lingeling ayu were winners in the Crafted and Application Tracks in SAT Competition 2014, respectively. We cannot exactly compare the performances between the VBS and the best solvers, because they were not tested in the same execution environment. However, our results have some penalties, because they were tested by running 15 solvers simultaneously on WS which have 24 cores with hyperthreading (12 physical cores), to reduce the experimental time, thus performance of solvers were deteriorated. Results in Table 6.1 were performed on WS using only single core. When we compare results of VSIDS, TBVSIDS2, and CHB between Table 6.1 and Figure 5.7, results in Figure 5.7 solved 65 more instances than those in Table 6.1. Despite of the deterioration, VBS from branching heuristics outperforms CSS in Table 5.2. We are comparing performances of VBS and CSS and both of them are virtual solvers. VBS outperforms CSS and we can reach to VBS by including a hybrid branching heuristic in a solver. Further, this solver is single-solver, thus it can be improved continuously.

Next, we provide list of features extracted from SAT instances for constructing a random forest model here.

- **Size:**
  - **1. Number of clauses**: clauses
  - **2. Number of variables**: vars

  - **3. Ratio**: vars/clauses
- **Variable-Clause Graph:**
  - **4-8. Features of variable nodes degree**: mean, variation coefficient, min, max and entropy.
  - **9-13. Features of clause nodes degree**: mean, variation coefficient, min, max and entropy.
- **Variable Graph:**
  - **14-18. Features of nodes degree**: mean, variation coefficient, min, max and entropy.
  - **19-23. Features of diameters**: mean, variation coefficient, min, max and entropy.
- $\boldsymbol{F_{13}}$: Features of Size and Variable-Clause Graph.
- $\boldsymbol{F_{23}}$: All features of abovementioned.

All features abovementioned can be extracted without a specific algorithm. We excluded algorithm-dependent features such as features after simplification methods or several values obtained during five minutes of initial search. We considered using algorithm-dependent features is not desirable, because these features might be useless when an algorithm changed.

The variable-clause graph $VCG = (V_1, V_2, E)$ is a bipatite graph, where V_1 is a set of variables, V_2 is a set of clauses, and E is a set of edges. An edge between a varible $v_i \in V_1$ and a clause $C_j \in V_2$ exists only when $v_i \in C_j$. The variable graph $VG = (V, E)$ is constructed based on each clause. Let L_i is a list of clauses including a variable v_i. An edge between variables v_i and v_j exists only if $L_i \cap L_j \neq \phi$. Figure 5.9 shows an example of VCG and VG. A number at upperright of each node in a VG indicates a diameter.

We defined two feature sets $\boldsymbol{F_{13}}$ and $\boldsymbol{F_{23}}$ those are used in our experiments. $\boldsymbol{F_{13}}$ can be extracted in a reletively short timeframe for existing SAT benchmarks. $\boldsymbol{F_{23}}$, on the other hand, cannot be applied to all instances. Several instances are too large to extract $\boldsymbol{F_{23}}$. We will apply random sampling method for approximate extraction of $\boldsymbol{F_{23}}$.

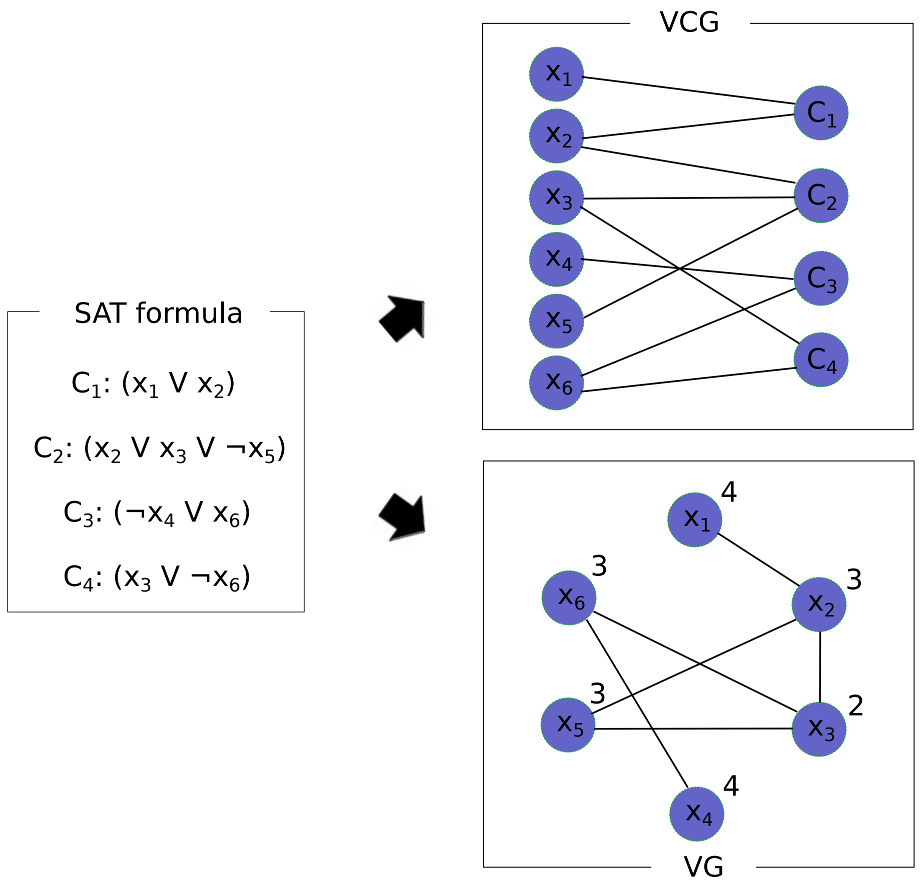


Fig. 5.9: An example of VCG and VG for a given SAT formula.

5.3.2 Experimental results - 13 features

In this section, we find the best hybrid model by activating subset of $\boldsymbol{F_{13}}$. Total feature extraction time of $\boldsymbol{F_{13}}$ against 1400 benchmarks from SAT competitions was under 1,000 s. As our model in a SAT solver is used as a preprocessing before search, rapid feature extraction is important.

There exist 8192 subsets of $\boldsymbol{F_{13}}$, and construcing each random forest model take about 10 s on WS. Therefore, constructing all possible combinations of 8192 models can be performed in a day. This a not a short time, but we performed all 8192 models for the analysis and comparison with results of $\boldsymbol{F_{23}}$.

Models are obtained based on features of SAT instances and running times for eight branching heuristics on Glucose 3.0. Instances unsolved by any of branching heuristics were excluded. Each of solved instance was labeled with a branching heuristic that showed the shortest running time. Partial results are illustrated in Tables 5.3, 5.4, and 5.5. Each table shows the test results through several training data. Each number in parenthesis at column 1 and row 1 indicates the number of training data and benchmarks, respectively. For example, in Row 7 of each table, the model was trained on all benchmarks except for benckmarks of Crafted Track in SAT Competition 2014, total of 697 instances and tested by each track. The results in Table 5.3 were obtained using all features in $\boldsymbol{F_{13}}$. When the performances of classifiers were evaluated by k-fold cross validation (Training data: ÂX | ĈX, Test data: AX | CX, where X = 14 | 15 | 16), the classifier in Table 5.4 showed the best performance, but surprisingly, used only one feature. The model in Table 5.5 was selected by our objective function f stated below, where X is one of the test data (AX or CX), and $N(X, Y)$ is the number of solved instances in test dataset Y when the model was trained by dataset X.

$$minimize\ f \tag{5.1}$$

$$f = A - B \tag{5.2}$$

$$A = \sum(N(X, X) - \alpha \times N(\hat{X}, X)) \tag{5.3}$$

$$B = \sum\sum(N(\hat{X}, Y)) \tag{5.4}$$

We explain the concept of our formulas. When the training data is used as the test data, i.e.,$N(X, X)$, the performance is high, but the performance of $N(\hat{X}, X)$ is low because it is trained exactly without the test data. Therefore, we seek to mimimize the gap between $N(X, X)$ and $N(\hat{X}, X)$ to achieve a good model. We also desire to reduce the gaps between the VBS results and the sum of $N(\hat{X}, X)$. Because the VBS results are fixed, they are excluded from the formulas. To combine these two ideas, a parameter α is added in Formula 5.3. Here we set $\alpha = 2$.

We named the classifiers in Tables 5.3, 5.4, and 5.5 as *all*, *k-fold*, and *f*, respectively. Let n be the number of solved instances when all instances were used as the training data and evaluated on themselves. We calculated correlation coefficients using 8192 results. The correlation coefficient of the performances between *f* and n was -0.59. Therefore, the per-

Table 5.3: Test results with several training datasets using all 13 features. Columns: Training data. Rows: Test data. C: Crafted Track and A: Application Track. $\hat{A}X$ = all - AX. $\hat{C}X$ = all - CX, where X = 14 | 15 | 16.

	C14 (300)	A14 (300)	A15 (300)	C16 (200)	A16 (300)	all (1400)
C14 (214)	**204**	198	244	29	137	812
A14 (231)	174	**215**	244	39	139	811
A15 (261)	159	212	**251**	42	137	801
C16 (65)	152	204	224	**65**	130	775
A16 (157)	169	206	240	35	**151**	801
$\hat{C}$14 (714)	**167**	217	249	61	143	**837**
$\hat{A}$14 (697)	198	**211**	247	62	143	**861**
$\hat{A}$15 (667)	201	213	**245**	63	145	**867**
$\hat{C}$16 (863)	204	214	246	**38**	142	**844**
$\hat{A}$16 (771)	201	213	249	63	**139**	**865**
all (928)	202	213	247	63	143	868

Table 5.4: Test results with several training datasets using only one feature in a variable-clause graph (variable nodes: max)

	C14 (300)	A14 (300)	A15 (300)	C16 (200)	A16 (300)	all (1400)
C14	**197**	201	234	23	133	788
A14	159	**214**	241	43	136	793
A15	194	214	**249**	36	138	831
C16	139	180	199	**63**	125	706
A16	182	207	236	36	**140**	801
$\hat{C}$14	**180**	218	247	56	142	**843**
$\hat{A}$14	196	**211**	245	56	139	**847**
$\hat{A}$15	196	215	**246**	56	142	**855**
$\hat{C}$16	202	216	247	**51**	142	**858**
$\hat{A}$16	193	215	248	56	**137**	**849**
all	196	214	244	56	139	849

formance of a single solver could be improved by minimizing *f*. The correlation coefficient between *k-fold* and n was 0.11, too low to claim a relation between these performances.

We further considered the expandability of our model. If a model is trained by results from a solver and show its efficiency in other solvers, we might insist a model is widely applicable. To achive this, we trained our model using the benchmarks results on Glucose and tested a model using another SAT solver, abcdSAT [107], which was the winner in the Main Track of SAT-Race 2015. The performance differences between Glucose and abcdSAT are proved quite different; out of 1400 instances, 169 instances were solved by

Table 5.5: Test results with several training datasets using seven features in a variable-clause graph (vars, clauses, vars/clauses, variable nodes: variation coefficient, min, max, and entropy)

	C14 (300)	A14 (300)	A15 (300)	C16 (200)	A16 (300)	all (1400)
C14	**202**	199	244	37	132	814
A14	174	**217**	251	38	134	814
A15	157	211	**252**	39	139	798
C16	160	199	214	**65**	116	754
A16	178	204	239	39	**144**	804
Ĉ14	**177**	216	251	62	140	**846**
Â14	203	**212**	251	63	141	**870**
Â15	202	214	**249**	59	141	**865**
Ĉ16	204	215	249	**37**	142	**847**
Â16	204	215	250	63	**141**	**873**
all	203	216	249	63	143	874

Table 5.6: Solved instances from SAT Competitions using abcdSAT. Row Average denotes the expected average performance of abcdSAT when a branching heuristic is randomly selected from 8 different branching heuristics.

Solver	C14	A14	A15	C16	A16	all
VBS	180	237	267	46	153	883
VBS$_C$	178.5	236.5	266.5	45	151.5	878
Average	162.9	221.3	254.5	37.8	141	817.5
all	168	228	256	37	143	832
k-fold	170	229	259	35	146	839
f	169	227	254	34	143	827
f_r	169	231	266	41	146	853

one solver but not by the other. Thus, we can assess the expandability of our model by applying it to abcdSAT.

Tables 5.6 and 5.7 compare the results of different models in abcdSAT and Glucose, respectively. Our models were trained by the Glucose results and applied to both Glucose and abcdSAT. All of our models were performed reasonably in abcdSAT, although the improvements were smaller than those in Glucose. VBSs in Tables 5.6 and 5.7 were obtained based on results of abcdSAT and Glucose, respectively. On the other hand, VBS$_C$s were obtained by considering both results of abcdSAT and Glucose. Let B_a and B_G are sets of branching heuristics that can solve a SAT instance. If $B_a \cap B_G \neq \phi$, then a branching heuristic $b_i \in (B_a \cap B_G)$ is selected in VBS$_C$s. If $B_a \cap B_G = \phi$ and $B_a = \phi$, $B_G = \phi$, or $B_a \neq \phi$ and $B_G \neq \phi$ then a branching heuristic $b_i \in B_G$, $b_i \in B_a$, or $b_i \in$

Table 5.7: Solved instances from SAT Competitions using Glucose.

Solver	C14	A14	A15	C16	A16	all
VBS	214	231	261	65	157	928
VBS_C	212.5	230.5	260.5	64	155.5	923
Average	158.4	196.4	227.1	41.4	128.9	752.2
all	202	213	249	63	143	870
k-fold	196	214	244	56	137	847
f	203	216	249	63	143	874
f_r	205	215	249	62	139	870

$(B_a \cup B_G)$ is selected, respectively. A model cannot satisfy both VBSs. VBS_Cs indicate actual VBSs for a single model when we consider both solvers Glucose and abcdSAT. If the differences between VBSs and VBS_Cs are large, we cannot find an efficient model for both abcdSAT and Glucose. However, the gaps between them were quite small, thus we can find an efficient model. We also demonstrated a hypothetical model f_r which showed the best results when we consider the performances of both solvers. At this time, f_r was found by a brute-force approach, namely, by traversing all 8192 models. Therefore, the performances of f_r are the best results when we limited a feature space in $\boldsymbol{F_{13}}$. In the following sections, we propose a random sampling to extract more features in a short time, and test on $\boldsymbol{F_{23}}$ to obtain better results than those of f_r.

5.3.3 Random sampling

In previous section, we extracted $\boldsymbol{F_{13}}$ from each SAT formula, and they do not require sampling method because of the fast calculations. However, when we attempt to extract more complex features such as constructing and extracting features from a variable graph or a clause graph, they are time-consuming processes and even infeasible when the numbers of variables and clauses are very large. We considered that several features, such as average and entropy, will be conserved even when the computations are reduced by random sampling. As a preliminary, we tested the validity of random sampling in SAT formulas using $\boldsymbol{F_{13}}$. We considered if random sampling is valid in $\boldsymbol{F_{13}}$, it would also be useful for more complicated features. Therefore, experiments in this section assess the possibility of random sampling for feature extractions in SAT formulas.

Figure 5.10 provides the correlation coefficients of feature extraction between the orig-

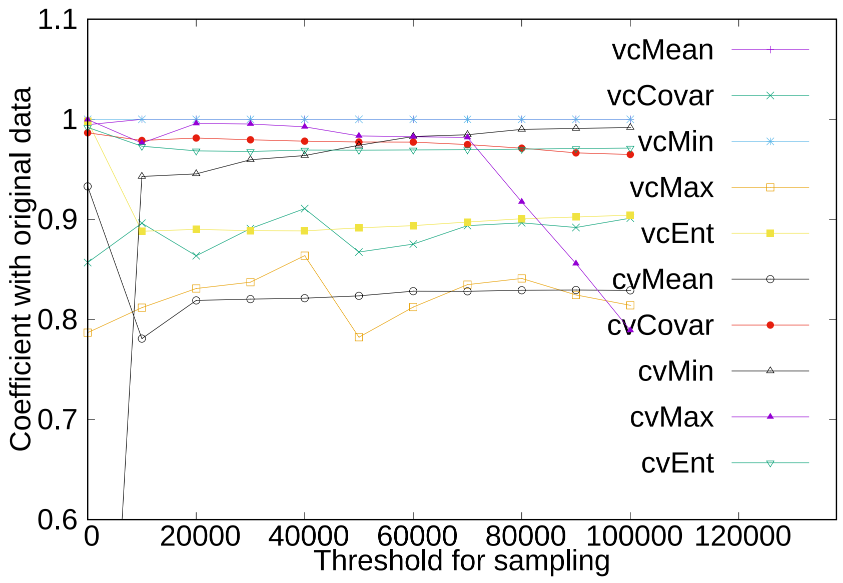


Fig. 5.10: Correlation of features between original and sampled formulas

inal formula and a randomly sampled formula using $\boldsymbol{F_{13}}$. Three features (vars, clauses, and vars/clauses) were excluded, because they do not require computational times. The sampling ratio was fixed at 0.1; meaning that 90% of the variables are removed from the original formula. If a clause includes several of the removed variables, then it shrinks, and if all variables of a clause are removed, then the clause itself is removed. The sampling method was applied when the formula included more than th variables, where th is a pre-set threshold. The threshold was imposed to alleviate concerns that several formulas are so small that sampling may hide their features. Each point was calculated 10 times and averaged.

Most of the coefficients were very high. The exception was cvMin at $th = 0$, because the average value of cvMin in the original formulas was very low (1.46). When applying the sampling method to all formulas, all of the cvMin values become 1.0, so the coefficient vanishes. The coefficients at vcMax became low sometimes because very few variables are connected to a large number of clauses, and if these variables are removed by random sampling, then the vcMax value becomes very small.

Overall, we concluded the performance of the sampling method was not bad, and it is worth while applying random method for feature extractions. Further studies of this approach might strengthen our intuition; in any case, the approach usefully reduces the time of extracting extra features.

5.3.4 Experimental results - 23 features

In this section, we expanded the feature set from $\boldsymbol{F_{13}}$ to $\boldsymbol{F_{23}}$ and applied the abovementioned random sampling method. The most time-consuming tasks were extracting the diameters. For a SAT formula with V nodes and E edges, the extraction of diameters using BFS from a variable graph constructed by linked list requires $O(VE)$ of runtime. From the relation $E = k \times V$ derived from the variable graph, the time for extracting the diameters is proportional to $k \times V^2$. By extracting only d_t diameters, where $d_t = N/(k \times V^2)$ and $N = 10^{10}$, we were able to extract $\boldsymbol{F_{23}}$ for 1,400 benchmarks in under 7 hours. The longest extraction time from all instances was 6 minutes, which is reasonable time for SAT solvers. Generally, features from variable graphs cannot be computed within a reasonable timeframe for several instances having a huge variable graph. When investing over 300 SAT formulas from the SAT Competition 2014 Application Track, 116 instances contained over 10^5 variables, and 26 or them contained over 10^6 variables.

$\boldsymbol{F_{23}}$ extracted by random sampling must then be validated. For this purpose, we constructed random forest models using $\boldsymbol{F_{23}}$ extracted by random sampling and benchmark results on Glucose. Figure 5.11 is a flow chart of the genetic algorithm which is designed to find an efficient random forest model for both Glucose and abcdSAT SAT solvers within a reasonable timeframe. Different $\boldsymbol{F_{23}}$s using different random seed numbers were used for training and evaluation phases, respectively. Different seeding make each random sampling select diffrent nodes from a variable graph producing different feature values. Therefore, we can experimentally demonstrate the usefulness of random sampling on $\boldsymbol{F_{23}}$ through our tests.

We adopted to apply the genetic algorithm for creating a random forest model for $\boldsymbol{F_{23}}$, because the search space of partial acvation of features can be represented by a boolean expression of size 2^{23}. We cannot attempt all cases this time in contrast with $\boldsymbol{F_{13}}$ experements. Figure 5.11 describes a flow chart for genetic algorithm, and we evaluated this with $N = 10$ and $K = 3$. The constructed random forest model is evaluated by a function g. Here, g is the squared sum of the differences between VBS and the solved instances in a model selected for Glucose and abcdSAT.

Within the feature space, we searched the optimum random forest model through the

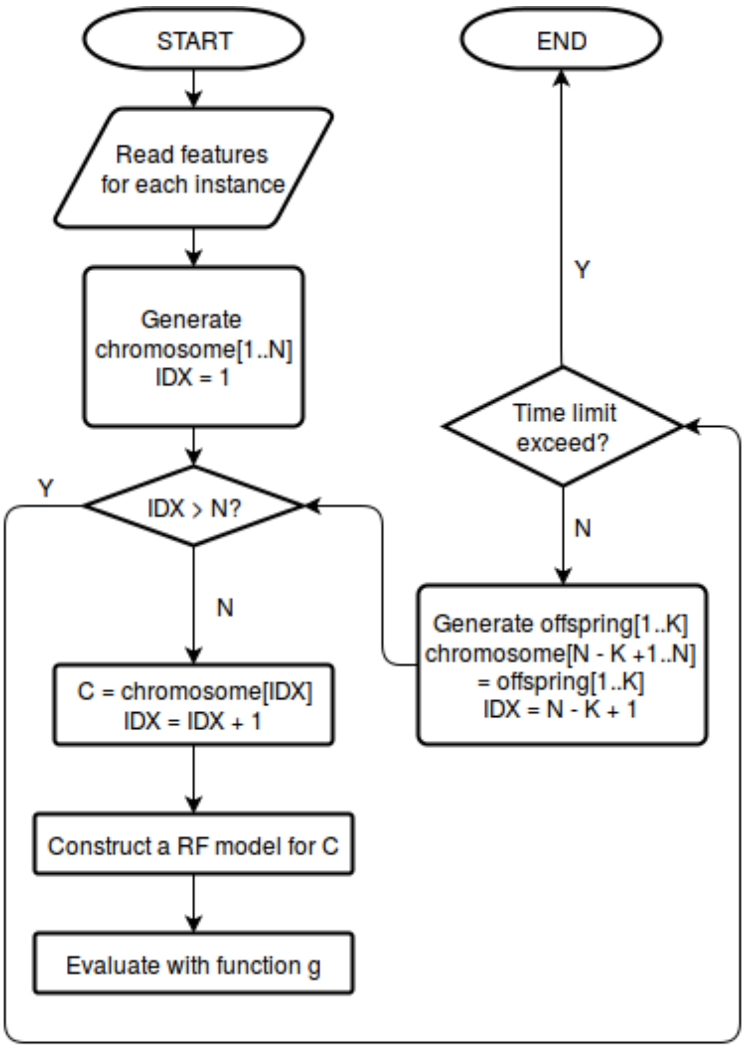

START
Read features for each instance
Generate chromosome[1..N] IDX = 1
IDX > N?
Y
N
C = chromosome[IDX] IDX = IDX + 1
Construct a RF model for C
Evaluate with function g
Time limit exceed?
Y
N
Generate offspring[1..K] chromosome[N - K +1..N] = offspring[1..K] IDX = N - K + 1
END

Fig. 5.11: Flow chart for generic algorithm for searching an efficient random forest model

above function g with the genetic algorithm (GA). We then tested the random search by constructing a point randomly in the feature space. The results are illustrated in Figure 5.12. The two lines at $Y = 853$ and $Y = 870$ indicate the performances of f_r in Tables 5.6 and 5.7. The GA outperformed the random search and stabilized after finding an optimal model within 8,000 s. These results are superior to those of f_r, confirming that expanding the features and extracting them within a reasonable timeframe improves the random forest model.

Finally, we summarize our results in Table 5.8. Numbers in a row Average indicate the expected numbers when a solver selects a branching heuristics randomly from 8 branching heuristics. VBS is generated by 8 branching heuristics. Each percentage with green color

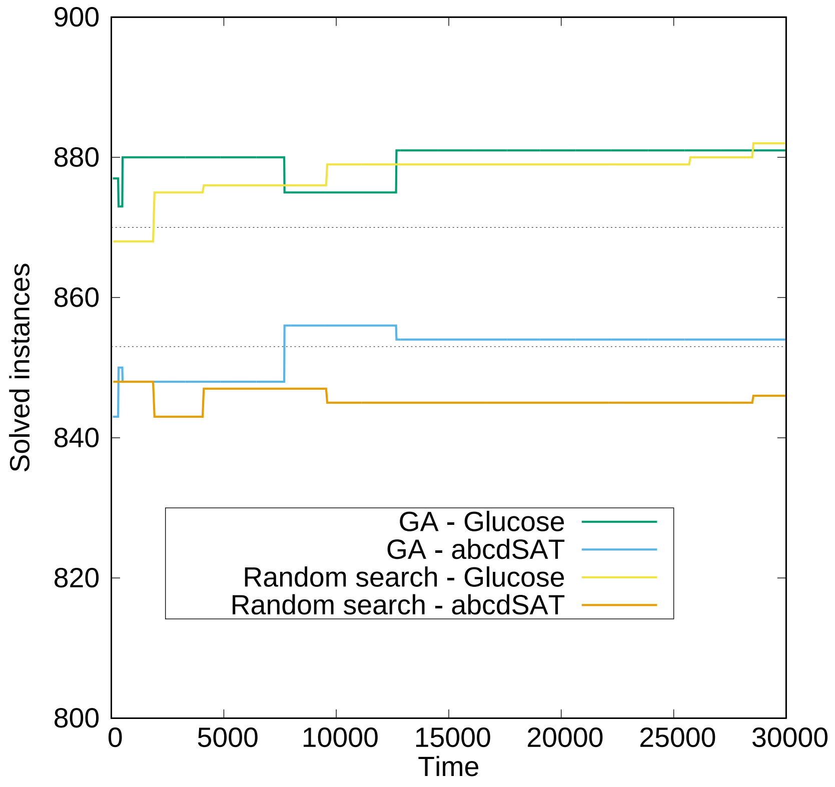


Fig. 5.12: Comparison of $\boldsymbol{F_{23}}$ models for genetic algorithm and random selection

indicates a solved ratio between VBS and average. Results of $\boldsymbol{F_{13}}$ and $\boldsymbol{F_{23}}$ were quite better than those in our static method descibed in Section 5.2. The best results were obtained by using $\boldsymbol{F_{23}}$ with random sampling within 2 hours. There still remains a gap between VBS and our results. However, we are considering that combining different hybrid models is better idea than improving performance of a single hybrid model. In other words, combining a hybrid branching heuristic and a hybrid restart might be better direction to improve the performance of SAT solvers instead of finding a better hybrid branching heuristic.

Table 5.8: Summary of hybrid branching heuristic models.

Model	Branching heuristics	CHB and TBVSIDS	8 branching heuristics		Average	VBS
	Features	Number of variables	$\boldsymbol{F_{13}}$	$\boldsymbol{F_{23}}$ (Random sampling)		
Solved	Glucose 3.0	798 (26.1%)	870 (67.0%)	**875 (69.9%)**	752.2 (0%)	928 (100%)
	abcdSAT	823 (8.4%)	853 (54.2%)	**856 (58.8%)**	817.5 (0%)	883 (100%)
Learning time for a model		-	1d	2h	-	-

Chapter 6

History map

In this chapter, we propose a concept of history map that would be applicable for the massively parallel environments in a combinatorial search. Firstly, we address backgrounds in a combinatorial search, raise problems in the current parallel SAT solvers, and explain the necessity of history map. Secondly, we introduce an application of a history map for the search diversification in a parallel SAT solver. Finally, we demonstrate our implimentation and evaluate an application through the benchmarks from SAT Competitions.

6.1 Backgrounds

The search complexities of the combinatorial algorithms are high dimensional. As we cannot traverse all the combinations to find a solution, solvers for constraint problems utilize algorithms to prune or select the areas of search space with estimations.

CPLEX [108], the most representative CP solver, utilizes branch and bound, evolutionary algorithm and many other heuristics.

Hypervolume estimation [109] calculates an approximate hypervolume for thee multi-objective optimization, because an accrate hypervolume calculation is time-consuming process.

SAT solvers learn clauses from conflicts, utilize branching heuristics and restart poliecies to prune, diversify and intensify searches.

The evolution of multicore hardware and cluster environments have effects on the progress of parallel solvers. There are two approaches for search space partitioning. Divide-and-conquer based parallel solvers allocate different subspace to each worker.

Portfolio-based parallel solvers do not divide search spaces. Workers in a portfolio parallel solver competitively and cooperatively find a solution. Hypothesize a solver can saves all archives from the past and havs a brilliant data structure to scan rapidly. A solver then can avoid duplicated search when it utilizes a portfolio approach. It would be able to construct a well-balanced search by allocating subspaces to workers adequately. This is an ideal hypothesis, but an efficient archive management would contribute improvements of SAT solvers.

Parallel SAT solvers share the learned clauses among workers. Clause sharings support to reduce duplicated works and induce unit propagations. However, finding an efficient clause sharing policy is very difficult. We raise several reasons to explain its difficulties.

- Potential number of learned clauses is too large to maintain them all. When a SAT formula have n variables, there exist ${}_nC_k \times 2^k$ possible k-length learned clauses.
- If we set a static method to share learned clauses based on for example their sizes of *LBD*s, the number of shared clauses fluctuate depending on SAT instances.
- We cannot estimate which learned clauses are useful in other workers. Shared clauses those are not used in a conflict analysis or a unit propagation for a long time increase the propagation time and lead inefficient search.

We explain more details for these three items.

Problems would be induced when a lot of short clauses are found. Figure 6.1 compares the results of the "esawn_uw3.debugged.cnf" instance from the SAT Competition 2014. Each curves are sorted by their run times. Of the 35 sequential solvers, 23 solved this instance within a specific time limit (most under 500 s); however, only four out of 15 parallel solvers were able to solve this instance. We are fairly certain that these results were due to memory limits. The "esawn_uw3.debugged.cnf" problem has over 10,000,000 variables and more than 50,000,000 clauses in its original SAT formula. We can learn from these results that learned clauses must be shared carefully due to memory limits.

ManySAT [67] is a good example for a second item. It adjusts sharing ratios dynamically but is too expensive, because it adjusts all pairwise ratios with an estimation of the quality of each learned clause. Measuring the qualities of learned clauses is also difficult because $TRUE/FALSE$ allocation of each variable changes rapidly.

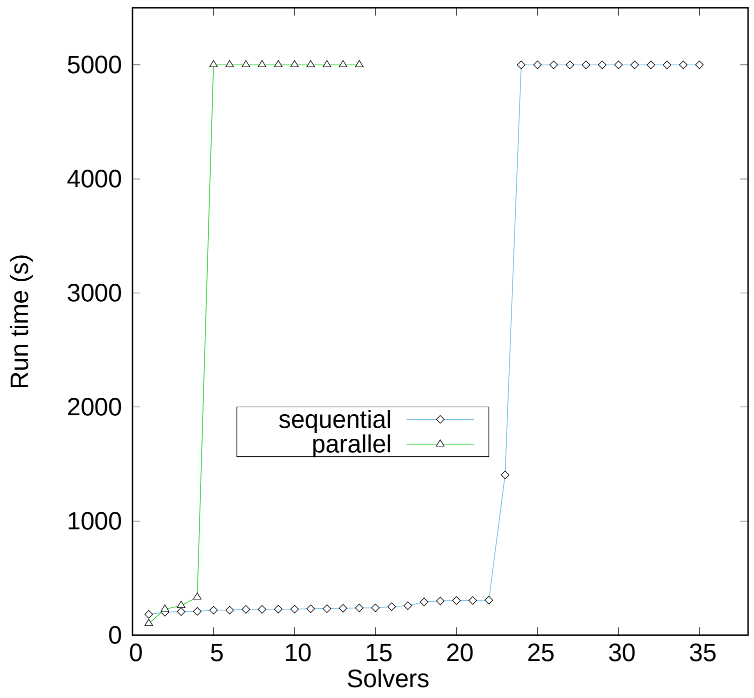


Fig. 6.1: Results of “esawn_uw3.debugged.cnf” for both sequential and parallel SAT solvers in SAT Competition 2014

As we already explained in 4.3.4, extracting more number of short clauses does not ensure the improvement of the solver efficiency. It indicates that a lot of learned clauses with low sizes or *LBD*s found by a worker w_i will not be used in w_j. Therefore, sharing them to another worker w_j might deteriorate the efficiency of w_j.

As abovementioned, there are a lot of potential learned clauses. Accumulated learned clauses reduce available resources for calculation and penalize propagation efficiency of a SAT solver. Therefore, the obtained clauses are periodically erased from learned clauses database to acquire resources. To share information efficiently without decaying old information, we propose an approximate history map (AHM). Details are discussed in the following section.

6.2 Concept of approximate history map (AHM)

Sharing information in a massively parallel environment must be done carefully. When a parallel SAT solver obtains a lot of learned clause, clause sharing among workers would require a large amount of memory even if only short clauses are shared. Importing clauses as much as a worker can from other workers is not desirable method. If the amount

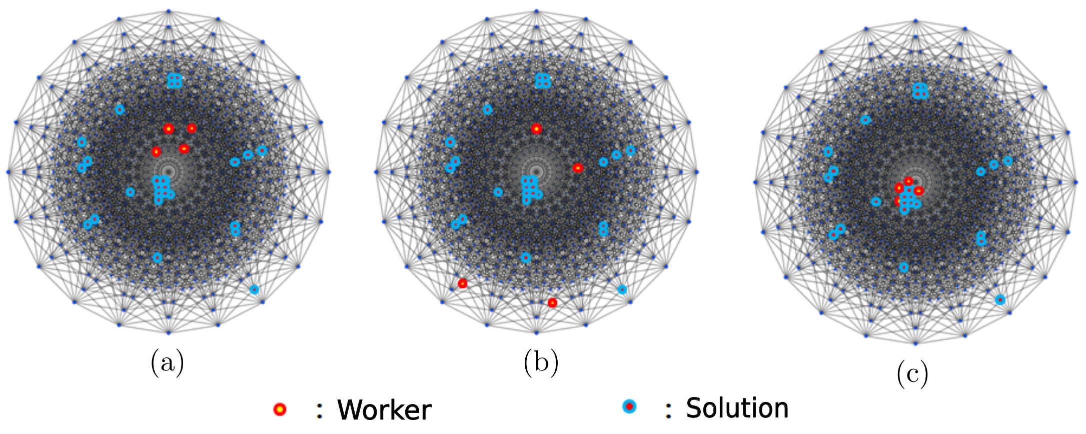


Fig. 6.2: Snapshot images of distributions of workers and solutions in the search spaces. Each search space is projected into the 10-cube space.

of information in a worker increases rapidly, a worker have to erase its learned clauses frequently to maintain sufficient memory and solver productivity. This implies that we have to restrict to share only clauses considered significant. However, determining which clause is helpful is impossible. For example, let us consider a clause C is discovered from a worker w_1 and shared to w_2. If C is used in a short time in w_2, we may consider clause sharing was useful. However, C might be not used for a long time, but eventually this clause may have an important role later. Therefore, many of solvers are sharing clauses with static methods such as their sizes or LBDs. When we use static methods, the amount of information flunctuate and may have become large. Anyway, when a number of workers increases, the sharing information would be increased and a woker have to erase them actively to maintain the rapid unit propagations. We considered this to be counterproductive; thus, we designed a method to reduce the amount of the information extremely to maintain the productivity of a worker in a parallel SAT solver.

Figure 6.2 illustrates the snapshot images of a parallel SAT solver in the middle of the search. Let us consider mapping of a search space of the n-variable SAT formula into a n-dimensional hypercube. Hypercubes in Figure 6.2 shows 10-cubes, thus this illustrates the hypothetical images of the projection of n-dimensional hypercube into 10-dimensional hypercube. Each vertex on a 10-cube represents several areas of the search space. Each circle with a red border represents the current position of each worker. Each circle with a

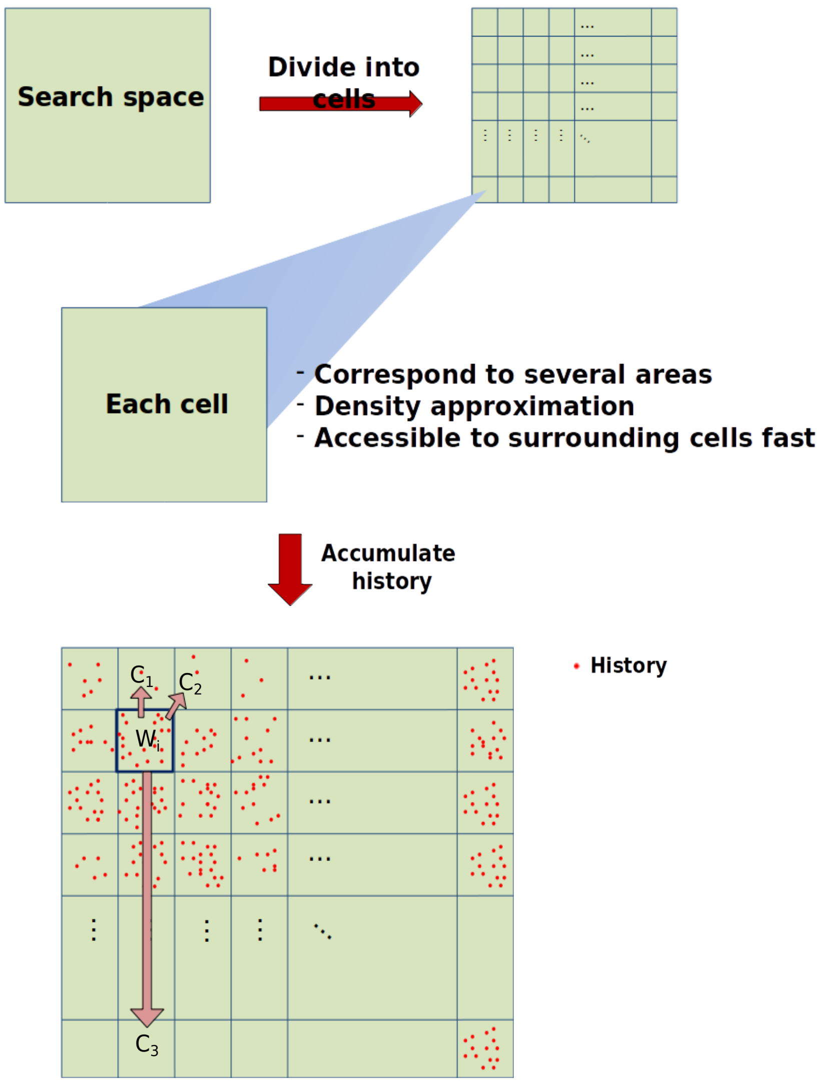


Fig. 6.3: Concept of an approximate history map on 2-dimensional space.

blue border represents a subspace containing solutions. Therefore, if a worker is located at a vertex without a blue circle, it cannot find a solution. Let us assume three 10-cubes in Figure 6.2 are representing the same SAT instance, thus distributions of blue circles in them are the same. We hypothesize workers are performing intensive search at (a) and (c) and diverse search at (b) in Figure 6.2. Workers in (a) are required to diversify search, because there is no solution around them. Intensive search near solutions such as (c) would be one ideal situation for parallel SAT solvers.

To estimate the distribution of solutions for the SAT formulas, we propose to design a density map integrated through the snapshots from each worker during searches. However, we cannot build a density map for solutions, because if a solver find a solution, search ends. This is nonsense. Therefore, we attempt to integrate no good areas instead. Accumulations of no good areas may indirectly construct a density map of solutions. Figure 6.3 describes a AHM concept. As a Search space for each SAT formula is too high-dimensional, we illustrated this on a plain for the simple explanation. A Search space is divided into several cells for the approximation. Therefore, each cell corresponds to several areas and approximates a density of these areas. Data structure is designed for a cell to access rapidly to its surrounding cells aimed for the dynamic migration into them. A solver accumulates “no good” histories. Red points in each cell indicate cumulated histories. AHM might be useful for several scenarios to diversify or intensify search. For example, if a worker W_i is curretly searching at a cell squared with thick lines in Figure 6.3. If a solver puts importance on search diversification, a cell C_3 would be a good candidate for the next search, because it has never been searched from any worker. Either C_1 or C_2 might be a good candidate, if a solver considers the balance between diversification and intensification, because both C_1 and C_2 are close from a current cell of W_i, and their densities look lower than those of their neighbors.

Th achive a AHM concept, we designed to reduce the amount of information of a history map so as to be shared and stored in parallel SAT solvers. In the first step, we compress the information to the extent possible. For a problem with n variables, we project a search space of an n-dimensional cube onto a one-dimensional integer array, i.e., the proposed AHM. Each integer value in the map corresponds to several areas in the search space. Next, we have to choose which information to share using a AHM. A number of modern SAT solvers are adopting the phase saving heuristic [60] for the search intensification. A solver with the phase saving saves the last *TRUE*/*FALSE* assignment for each variable in a phase saving array. Right after conflicts, this array indicates “no good” status. Therefore, we convert this status into an integer, which we refer to as the poliarity search space index (*PSSI*). Details are discussed in the following section. Calculating a *PSSI* requires $O(n)$, where n is the number of variables in a SAT formula. As an attempt to reduce time cost for *PSSI*, a solver only calculates the *PSSI*s on every k conflicts. Then, a point corresponding to a *PSSI* in a AHM is increased by 1.

Next, we define the density of a point in a map. Here the density of a point is the sum of the visited counts of the areas that correspond to that point. For example, consider a point P in a history map, where P corresponds to 10 areas v_1, v_2, ..., v_{10}. Let area set $A = \{v_3, v_5, v_{13}, v_3, v_6\}$. If a map is updated by A, then P is incremented four times, excluding v_{13}, which is not included in P. It is evident that the actually visited areas are the three areas of v_3, v_5, and v_6. Area v_3 is counted twice. Determining the actual density of a point would require significant time and map space. Thus, we approximate the density by simply incrementing regardless of duplications. This is a key concept for reducing the amount of information.

6.3 Conversion of current area into an index

In this section, we propose a method to convert the current area of the search space into an index to reduce the amount of information. We accumulate converted indexes into a history map, and share this map among workers to balance between diversification and intensification of serach. We especially concentrate on areas right after conflicts, and share these to other workers because revisiting this area in other workers is undesirable. If all learned clauses are shared among workers and not erased, then a solver is able to avoid revisits of nogood areas. However, a solver cannot share all clauses and clauses are erased periodically for its productivity. Therefore, a method to extremely condense the amount of information is required to share nogoods approximately.

6.3.1 Proposal of Polarities to Search Space Index (*PSSI*)

Many state-of-the-art solvers are using phase saving [60] array (PSA) to reuse its previous phase for intensive search. The PSA contains last *TRUE*/*FALSE* assignments of variables and is reused in decision heuristics to determine a polarity (*TRUE*/*FALSE*) of a selected variable. Therefore, PSA determines search direction and contains a nogood assignment shortly after conflicts. Phase saving worked efficiently in sequential solvers, because PSA has a strong relationship with learned clauses found from the current worker. However, parallel SAT solvers share learned clauses, and there are low relavances between a PSA and clauses imported from other workers.

We can anticipate the effects accompanied by small changes in a PSA. They might deteriorate the interplay between a PSA and learned clauses found from the current worker, but a solver would obtain search diversity and might have an opportunity to use diverse clauses exported. We adopt to share a PSA among workers to create a AHM in a parallel SAT solver.

We explain details for converting the current PSA into a *PSSI*. First, divide a variable set into k-blocks $(B_1, B_2, ..., B_k)$. Second, calculate the ratio $(r_1, r_2, ..., r_k)$ of variables currently allocated to *TRUE*, and divide the ratio into m sections uniformly, and each section have a value between 0 on the far left to m - 1 on the far right. For each block B_i, ratio is converted to integer b_i. For B_i, $b_i = p$ if $p/m \leq r_i < (p+1)/m$ where $p \in \{0, 1, ..., m-1\}$. After calculating each b_i, calculate a *PSSI* (Polarity Search Space Index).

$$PSSI = \textstyle\sum_{i=1}^{k} b_i \times m^{i-1}$$

Since the *PSSI* is now only an integer, we can then easily though roughly compare the areas in the search space among the workers. Let's explain this with a simple example. Consider a problem with n variables x_1, x_2, x_3 and x_n. Solve this problem using the parallel SAT solver with 2 workers w_1 and w_2. Let's call p_i the current PSA of w_i. If we simply calculate the hamming distance between p_i and p_j, it takes only $O(n)$ time. However, to compute the distance between workers, they have to be synchronized, and this method would be unwieldy when the number of workers is increased.
If a conflict occurs at $p_1 = \{0,\ 1,\ 1,\ ...,\ 0\}$, then all workers w_i in a solver want to avoid the p_1 status in the future. However, memoization for this needs a lot of memory, thus we reduce the amount of information with estimation.

In *PSSI*, for example, if we suppose $k = 4$ and $m = 2$, ratio(w_1) = (0.3, 0.7, 0.0, 1.0) and ratio(w_2) = (0.6, 0.7, 1.0, 0.2), then we get $PSSI_1 = (0 * 2^0 + 1 * 2 + 0 * 2^2 + 1 * 2^3)$ $= 10$ and $PSSI_2 = (1 * 2^0 + 1 * 2 + 1 * 2^2 + 0 * 2^3) = 7$. We can compare their areas using their *PSSI*s based on bitwise XOR. In this case $(0101 \veebar 1110) = 1011$, and we count the number of 1s and get distance = 3.

We can obtain distances among workers directly such as an example abovementioned, but direct comparisons of them require synchronization. Therefore, we considered indirect comparison of using history map would be an efficient method, because it also enable to take into account past *PSSI*s.

*PSSI*s have k-blocks, and this structure will help maintain intensification in search. We can easily modify fractional changes by picking a block and inverting each polarity in a PSA. Using this structure, a solver would change a PSA dynamically. Details of an application will be described in Section 6.4.

Currently we are dividing k-blocks by simply using indexes of variables, which means if there are kx variables in the original formula, we allocate $1 \sim x$ to block 1, $x + 1 \sim 2x$ to block 2, and so on. We adopted this policy to try to minimize sudden changes in the current area. We discuss this in the following section.

6.3.2 Block division Policy

It is difficult to determine how to divide a variable set into the blocks optimally. Using a community detection algorithm such as the Louvain method [10] would be a good idea, because when we pick a block representing a community and change their assignments in a PSA, these will have the fractional effects to entire search because a community might have only sparse connections to other communities. However, a solver have to construct a variable graph to build communities, and this requires a lot of time and sometimes infeasible for large instances. Therefore, we adopt a policy to divide blocks with their indexes based on our experiments. We came up with this idea, because applcation instances are hand-made and there exist biased structures based on the indexes of variables.

We performed 300 repeat tests for a benchmark to produce different solutions, and observed their polarity trends. The benchmark chosen was 002-80-8.cnf from application instances in the SAT competition 2014, because a solution for this benchmark can be rapidly found most of the time. For each test we obtained a solution and checked its final polarities. Figure 6.4 gives the polarity distribution counts for each variable from 300 repeated tests. Each point indicates the number of *TRUE*s assigned in the solutions. Therefore, variables on the lines of $y = 0$ or $y = 300$ almost certainly are backbones [110] in a SAT formula because they showed the same assignment in all 300 tests. Figure 6.4.(a) was performed through an original SAT formula. We considered there is a high probability of having strong relations among proximate variables by index. For example, in Figure 6.4.(a) variables having an index between 0 and 2,500 have similar *TRUE* assigned numbers. We can also find a group of variables having under 30 *TRUE* assigned numbers at indexes between 9,500 to 13,300 relatively frequently. From this experiment, we suspected there might exist some hidden structures based on the indexes of variables.

To ensure the polarity distiribution is not a solver-dependent but a structure-dependent, we transformed an original formula and results of them are illustrated in Figure 6.4.(b), (c) and (d). When a formula have n variables, we converted each literal l_i to $\neg l_i$, l_{n-i} and $\neg l_{n-i}$ in (b), (c) and (d), respectively. As shown in Figure 6.4.(b), (c) and (d), they showed the same polarity distributions, even though we transformed a formula. Therefore, we considered its distribution is a structure-dependent, and adopted to divide blocks using the index order based on these experiments.

6.3.3 Representability of *PSSI*

To construct a history map through the *PSSI* indexs, we have to check how well *PSSI*s represent the entire search space. If all the solutions found have the same *PSSI* value, applying *PSSI* for a history map would be unsuitable.

We performed repeat tests for several benchmarks to observe their *PSSI* distribution. We chose 4 satisfiable benchmarks from application instances in the SAT Competitions 2014 that can relatively easily find a solution in a reasonable time. The lists are 008-80-8.cnf, 002-80-8.cnf, 004-80-8.cnf and 004-22-160.cnf. We did 20 repeat tests per benchmark with a time limit of 3,600 s, totaling 80 repeat tests. Repeast tests were performed using our parallel SAT solver ParaGluminisat. Normally, workers in parallel SAT solvers are asynchronized in attempts to maximize their performances. Therefore, they do not ensure a solution reproducibility, and produce a different solution for each execution.

For 10 tests a solver reached a time limit. For 70 tests solutions were obtained, and we checked their *PSSI*s. A *PSSI* is calculated under the condition of k = 10 and m = 4, so there exists 4^{10} different *PSSI*s. There were 67 different *PSSI*s found in 70 solutions. Let us assume the *PSSI* space is ideal that solutions are distributed uniformly on this space. Then, the possibility of 67 kinds from 70 *PSSI*s is extremely low. Equation 6.1 describes possibility of over 68 kinds from 70 *PSSI*s.

$$\frac{\binom{4^{10}}{70} \times 70! + \binom{4^{10}}{69} \times 69! \times \binom{70}{2} + \binom{4^{10}}{68} \times \left(68! \times \binom{70}{3} + 66! \times \binom{70}{2} \times \binom{68}{2}^2\right)}{4^{700}} \simeq 0.9999999982 \tag{6.1}$$

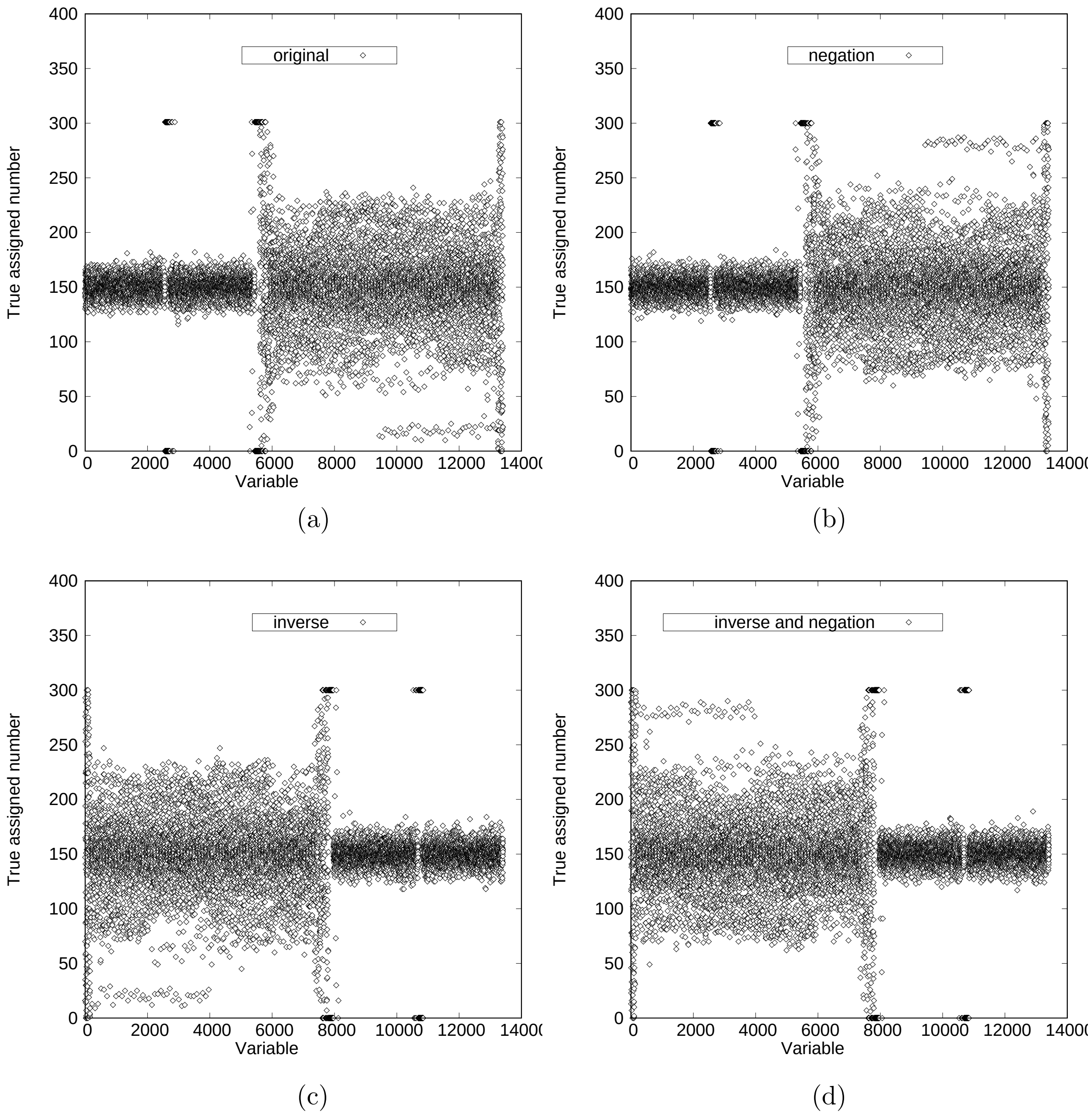


Fig. 6.4: Polarity distributions of each variable from 300 repeat tests. Benchmark: 002-80-8.cnf.

Therefore, we can assume that a *PSSI* space does not represent uniform distribution of solutions. However, at least these results show us that solutions for a SAT formula might be found in many different areas in the search space, and these areas might be distributed in some extent through the *PSSI*s on a history map.

6.4 Application of AHM - SaSS heuristic

We proposed the *PSSI* for constructing a AHM with reduction of the amount of information. In this section, we propose an application of AHM as an attempt to diversify search in parallel SAT solvers.

Many state-of-the-art parallel SAT solvers construct a portfolio to enhance their performances. A diversified portfolio may increase search diversification of the solver, but we concern its demerit. Details will be discussed in the following section. Several solvers have tried to compare each pair of workers directly, but these might work inefficiently in massively parallel environments.

We considered our AHM could be utilized to diversify the search. Our main idea is to avoid the areas frequently visited, and to walk towards the sparsely visited areas. Each cell in a AHM represents several areas and is indexed by its *PSSI*. A cell has a counter and is incremented by one when a worker's PSA corresponds to *PSSI* of this cell. Each cell is able to access its neighbors with bitwise operations. Therefore, we considered a solver can walk from the current area to the sparsely visited area dinamically with a AHM, and we propose the SaSS (Sparsely visited area walking on Search Space) heuristic.

Algorithm 6 and 7 describe the pseudo-code of the SaSS, and we remark the target area. We adjust partial assignments in a PSA to move to the sparsely visited area, but a solver might fail to reach or stay there. For example, if a worker adjust a PSA by force at decision level d_i, then a lot of assignments were already performed at decesion level between 0 to $d_i - 1$. Therefore, several changes does not work immediately, and early assignments before adjusting a PSA might prevent to reach sparsely visited area by forcing different assignments. Therefore, we called the sparcely visited area chosen by SaSS heuristic as the target areas.

Algorithm 6 SaSS heuristic: *changeCurrentArea*()

```
1: if $conflicts \ \% \ interval == 0$ then
2:     $p := getCurrArea()$;
3:     $p\prime := updateHistoryMap(p)$;
4:     if $p \mathrel{!=} p\prime$ then
5:         $changeBlockPolarities(p,\ p\prime)$;
6:     end if
7: end if
```

Algorithm 7 SaSS heuristic: *updateHistoryMap*(p)

```
1: Input: PSSI $p$
2: Output: PSSI $p\prime$
3: $p\prime := p$;
4: historyMap[$p\prime$]++;
5: if $p\prime <$ c-threshold $\times$ thread number then
6:     return $p\prime$;
7: end if
8: $p\prime :=$ checkNearestAreas($p$, $d$);
9: return $p\prime$;
```

In Algorithm 6, after every *interval* conflicts (line 1), each worker calculates the current PSA as *PSSI* (line 2), updates the history map of the *PSSI*s (shared for all workers) and gets a target area as a *PSSI* (line 3). A worker do not perform this on every conflicts to maintain its productivity. If the target area is different from the current area (line 4), it changes polarities in a PSA to walk towards the target area (line 5). A block can be easily calculated by the bitwise XOR of p and $p\prime$ for a PSA adjustment. If the selected block is B_i, then each variable's polarity is allocated to *TRUE* b_i in m. In Algorithm 7, we get a target area using a history map. It updates history map (line 4), but does not change the area in the early stages to maintain intensive search (line 5-6). When the early stages end, it searchs the target area based on the current area (line 8). It checks areas within the hamming distance d from the current area, and the one with the minimal counter in the history map is picked as the target area.

6.5 ParaGluminisat Implementation

We describe our parallel SAT solver ParaGluminisat in this section. This solver is also used in Chapter 3. First we remark a solver used in Chapter 3. In the next, we explain several options to enhance the solver performance.

We implemented ParaGluminisat based on Glueminisat [95]. Algorithm 8 describes pseudo code of ParaGlueminisat. ParaGluminisat was parallelized with the OpenMP library. Each worker solves a SAT formula competitively, and they share information in the shared memory class. We adopted data structure in ManySAT [66] for clause sharing. A solver shares learned clauses with a two-dimensional array with head and tail pointer to excess to the array exclusively without mutual exclusion process to achieve efficient clause sharing process. A clause c is exported to a share memory class, when its *LBD* is under L (line 11). After exporting, a worker imports clauses (line 12). When importing, propagations are performed through imported clauses. Therefore, a worker would find a solution or prove unsatisfiability while importing clauses (line 13). We apply the SaSS heuristic on every *interval* conflicts (line 10). The SaSS heuristic is performed with pthread_mutex_lock and pthread_mutex_unlock for mutual exclusion. However, mutual exclusion does not deteriorate the performance of the solver a lot, because the SaSS is performed only once in *interval* conflicts, and the process between pthread_mutex_lock and pthread_mutex_unlock is performed rapidly.

Algorithm 8 ParaGluminisat implementation.

```
 1: omp_set_num_threads(T);
 2: #pragma omp parallel
 3: {
 4:     int t = omp_get_thread_num();
 5:     workers[t].solve();
 6: }
 7: ...
 8: solve()
 9: {
10:     if ($conflicts$ % $interval$ == 0) SaSS();
11:     if($LBD(c) \leq L$) exportClause($c$);
12:     int answer = importClauses();
13:     if(answer != undefined) return answer;
14: ...
15: }
```

We utilized a ParaGluminisat without the SaSS in Chapter 3. Each worker shuffles indexes of variables for search diversification.

In this chapter, we implemented several addtional ideas to enhance the performance of ParaGluminisat. We applied these because we want to evaluate the SaSS on an efficient solver. We considered, if a solver is inefficient, then many approaches easily improve it.

We divided workers into two groups. Their is an *use_minisat* option in a Glueminsat. Glueminisat's base solver is MiniSat and the policies of MiniSat are easily applied with this option. We activated this option in the halves of threads, and inactivated the rest of them. We applied this because state-of-the-art solvers with a *LBD* measure with a rapid restart policy show their efficiencies for unsatisfiable instances. However, MiniSat still shows its efficiencies for satisfiable instances. To achieve intensive search, we only share learned clauses among workers with the same option. We set the halves of threads with no activation of *use_minisat* option to share clauses within $LBD = 2$. The other threads shared only unit clauses, because we considered search diversification is very important to solve the satisfiable instances and MiniSat solves them well. It is natural that we share unit clauses. Therefore, if we separate this from sharing policies, we can assume workers with no activation of the *use_minisat* cooperate, and each worker using *use_minisat* works independently.

ParaGluminisat is designed for applying many number of workers, not for specific number of workers. In contrast to general portfolio-based parallel solver, ParaGluminisat does not allocate different policies for each worker. Only a different seed value is allocated for each worker to diversify the order of variables' indexes, and workers are divided into two groups by activating and inactivating a *use_minisat* option, respectively. There are no differences except for components mentioned here. If each solver has a different restart interval, learning scheme and decision heuristic, diversity will arise. However, this might lead too many learned clauses, and sharing these might deteriorate each worker's intensity. Further, optimizing the best portfolio for workers would be difficult when the number of workers increased.

6.6 Experimental results

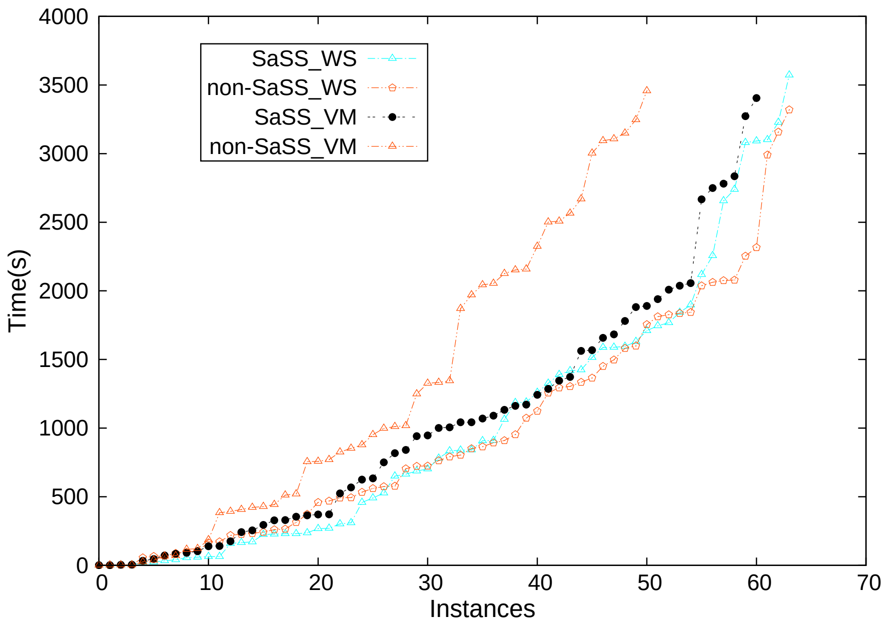
In this section, we present an experimental evaluation of the SaSS heuristic. SaSS is implemented in our parallel solver ParaGluminisat and evaluated through the number of solved instances and their runtimes for both satisfiable/unsatisfiable. Generally, parallel SAT solvers are evaluated with 3 repeat test, because they are not deterministic but opportunistic. Therefore, we evaluated the SaSS on two different environments WS and VM for two different numbers of workers.

Parallel track benchmarks from SAT-Race 2015 were used for the evaluation. As a default setting, each worker updates a history map in a shared memory using the *PSSI*s on every 50,000 conflicts. *PSSI* is calculated under $k = 10$, $m = 4$, and this indicates we have a total of 4^{10} different *PSSI*s. SaSS only checks proximate areas having a hamming distance of one from the current cell. In order to assess scalability, we tested the SaSS using both 12 workers and 64 workers. We want to compare results between SaSS and non-SaSS, but also between 12 workers and 64 workers. For the comparison, we set the CPU time limit to 3600$\times$ (number of workers) seconds. WS and VM only have 24 and 32 threads including hyper-threading, respectively. Therefore, the real time limitation is around 3600 seconds for 12 workers, but for 64 workers, the time limitation is around 9,600 seconds in WS and 6,400 seconds in VM.

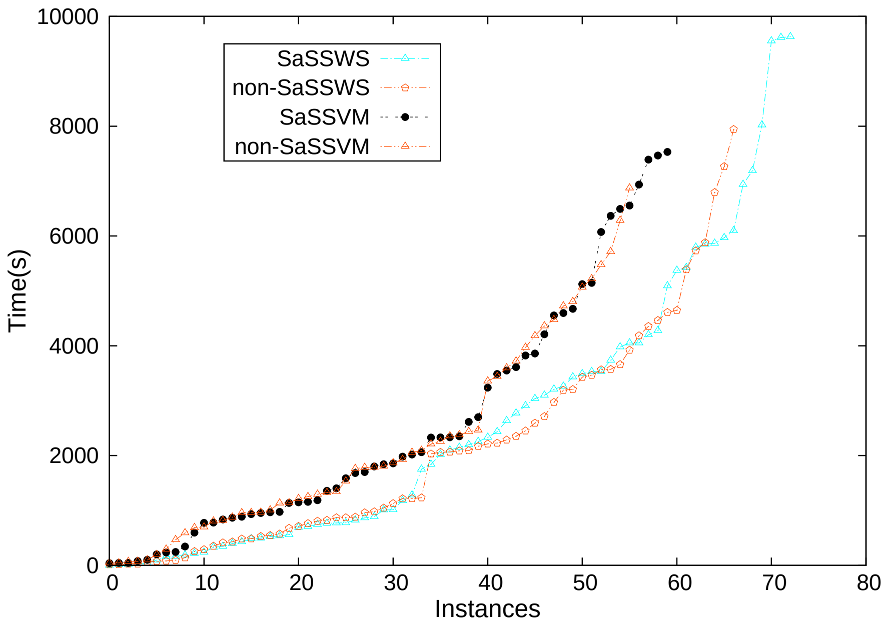
Figure 6.5 and Table 6.1 show the results of SaSS and non-SaSS. VM is based on VMWare, and it's results were worse than those of WS. Figure 6.5 compares the number of solved instances on both 12 and 64 workers. Table 6.1 summarizes the solved instances for each case. In summary, SaSS solved five more instances from the 100 instances on average, most of which were satisfiable instances. When we compare results between SaSS and non-SaSS with 4 different cases, the performances of SaSS were not better than those of non-SaSS at WS with 12 workers. Results of SaSS were better than those of non-SaSS in the rest of cases. This is why several repeat tests are required to evaluate parallel SAT solvers. If we simply count the number of solved instances, we might insist SaSS showed better performances than non-SaSS for satisfiable instances, but for unsatisfiable instances, SaSS does not deteriorate results, but we cannot claim it brought better results.

To observe more details, we displayed the results using a scatter plot in Figure 6.6. We divided regions to compare the speedup, and the outline of each region is described in Figure 6.7. In addition, we counted the solved instances in each region, and the results are shown in Table 6.3. R1, R3, and R5 represent regions in which non-SaSS surpasses SaSS. R2, R4, and R6, on the other hand, represent regions in which SaSS surpasses non-SaSS. R1 and R2 are divided by the $y = x$ line. In regions R3 and R4, areas satisfying both equations of $y \leq 1.3x$ and $x \leq 1.3y$ are excluded to observe speedup more strictly by excluding the region where the difference ratio between SaSS and non-SaSS is less than 30%. Regions R5 and R6 compare speedup by counting instances solved by only one side. In all cases, SaSS surpasses non-SaSS, as is shown in Table 6.3. We conclude the SaSS heuristic using a history map showed its efficiency by avoiding frequently visited areas and visiting sparsely visited areas. This approach could be applied for massively parallel environments because sharing a history map among workers requires only tiny time and space.

We obtained the best results on WS with 64 workers and 73 instances were solved. This



(a) 12 workers



(b) 64 workers

Fig. 6.5: The time required to solve benchmarks within a time limit of 3600×*worker number*. Limitation is for CPU time, not real time. Measure time (Y-axis) is real time.

Table 6.1: Solved instances with 100 benchmarks of SAT-Race 2015. S: satisfiable instances, U: unsatisfiable instances.

Env, Method	WS				VM			
	SaSS		non-SaSS		SaSS		non-SaSS	
Workers	12	64	12	64	12	64	12	64
S	42	49	42	44	39	41	33	37
U	22	24	22	23	22	19	18	19

Table 6.2: Solved instances with 100 benchmarks of SAT-Race 2015 on WS with 64 workers. S: satisfiable instances, U: unsatisfiable instances. AVT: average time (s)

	U	S	AVT
non-SaSS, $LBD = 2$	23	44	1718
SaSS, $LBD = 2$	24	49	1691
non-SaSS, $LBD = 8$	30	44	1390
SaSS, $LBD = 8$	31	48	1310

is a tie-score with 2_nd solver in a SAT-Race 2015, but a top solver solved 78 instances. We considered our clause sharing policy is too limited compared with the policies in other solvers. We shared clauses under $LBD = 2$ above, because sharing a lot of clauses does not scale when the number of workers increases. However, only 64 workers are required for the comparison, and we performed extra tests by expanding $LBD = 8$ for clause sharing and results are shown in Table 6.2. Now we obtained better results than a top solver in a SAT-Race 2015, and this is achived by applying a SaSS. Therefore, we experimentally demonstrated the SaSS is a valid in a state-of-the-art parallel SAT solver. As we abovementioned, sharing more clauses is important for unsatisfiable instances. Several unsatisfiable instances were solved by sharing more clauses, but there were little change for satisfiable instances. The SaSS heuristic mainly improved results for satisfiable instances. It is well known factor that clause sharing is not important for solving satisfiable instnaces, thus, we can conclude base on our experiments that SaSS is valid for solving satisfiable instances efficiently regardless of the change of the clause sharing policy.

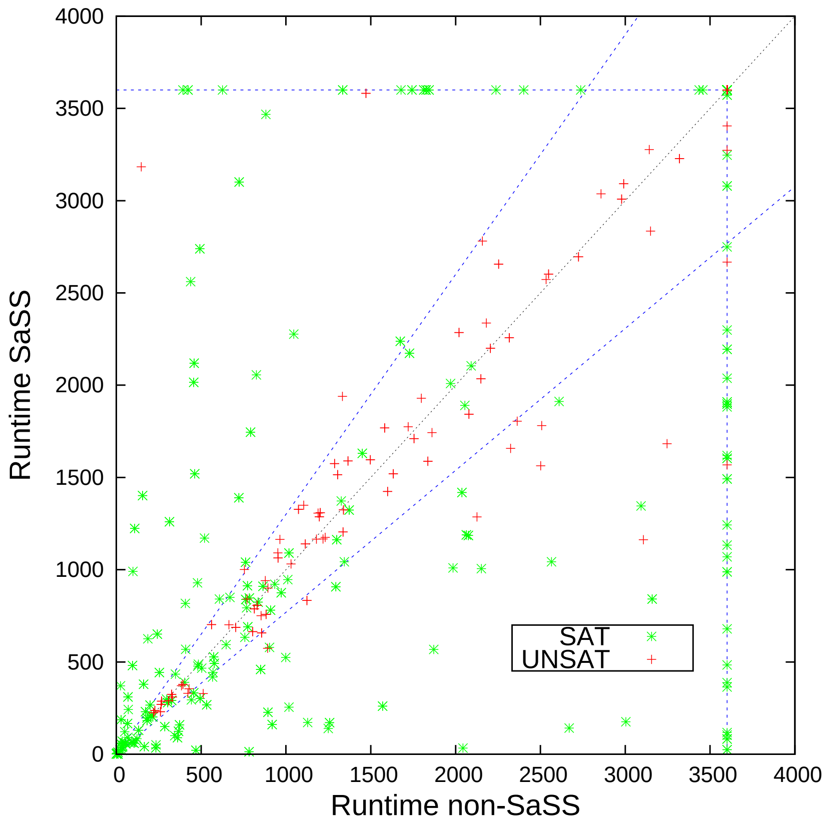


Fig. 6.6: Scatter plot: Comparison of runtimes with and without SaSS.

Table 6.3: Solved instances at each region in Fig 6.6

	satisfiable	unsatisfiable
R1 : R2	82 : 99	40 : 46
R3 : R4	51 : 68	4 : 13
R5 : R6	14 : 26	1 : 5

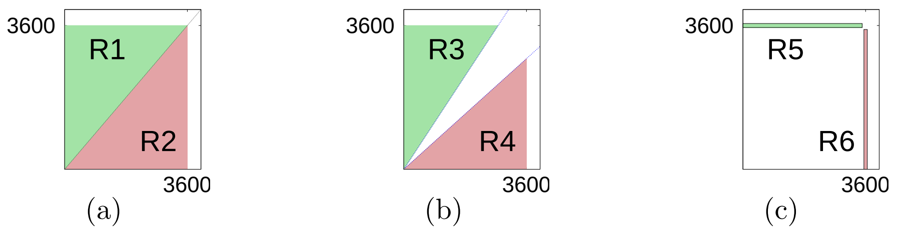


Fig. 6.7: Outline of each region in table 6.3.

Chapter 7

Other proposals

This chapter provides several SAT related ideas for future works; however those are not implemented or require some improvements.

7.1 Proposals for SaSS

In Section 6.4, we proposed the SaSS heuristic as an application for AHM. The motivation of SaSS is to move to sparsely visited areas dynamically while avoiding frequently visited areas, because we attempted to integrate nogood situations. Therefore, we can expect two key advantages by applying SaSS. First, SaSS would lead search diversification. Second, a solver might escape from areas having low probabilities for finding a model. We considered that additional diversification of the search would lead to better results for satisfiable instances, and we evaluated the proposed SaSS experimentally and obtained better results than non-SaSS. We explained AHM is managed by cell structure in Figure 6.3. In SaSS, a solver compares densities of cells, the current cell and its surroundings, using a history map. Then, a solver selects a cell having the lowest density.

In the following sections, we suggest two approaches that might enhance the performance of SaSS.

7.1.1 Biased random walk

SaSS selects a cell of areas with the lowest density as a target cell to move deterministically. We considered that if we select the target cell using a biased random walk, search diversification of SaSS would be strengthen.

Let C_i be a cell in a history map and $d(C_i)$ be the value of cell C_i, i.e., the density of the given point. Instead of selecting the cell with the lowest $d(C_i)$, we propose to select C_i with $prob(C_i)$ calculated as follows.

$$S = \text{A set of current cell and its surroundings} \tag{7.1}$$

$$r(C_i) = \begin{cases} \frac{1}{d(C_i)}, & \text{if} d(C_i)! = 0 \\ T(T > 1), & \text{otherwise} \end{cases} \tag{7.2}$$

$$SUM = \sum_{i \in S} r(C_i) \tag{7.3}$$

$$prob(C_i) = \frac{r(C_i)}{SUM} \tag{7.4}$$

We allocate the probability of each cell in proportion to the inverse number of its density, except when its density is zero. Therefore, we add T, which is greater than one, to balance the selection of a low-density area with high probability. For example, consider a cell C_1 in a history map with surrounding cells C_2, C_3, and C_4. If $d(C_1) = 1$, $d(C_2) = 2$, $d(C_3) = 3$, and $d(C_4) = 0$, then C_4 is selected deterministically via SaSS; however, in biased random walk, each of the surrounding cells can be selected even though C_4 would have the highest probability.

7.1.2 Density evaluation

SaSS selects a cell of areas with the lowest density regardless of the density value. If a cell in a history map represents k areas and the density is $0.01k$, then we can consider most areas in a cell are still not visited. In this situation, it would better to stay in the current cell instead of moving to a cell with lower density.

Therefore, we considered several factors to evaluate a current cell more appropriately.

First, we considered the relationship between the size of the areas that a cell represents and the density value. Here, we use the coupon collector's problem [111] to evaluate whether the density value is too low to change the areas. If all areas have the same probability of a visit, and a cell corresponds to m areas, then the expected time T to visit all m areas is evaluated as follows.

$$E(T) = \sum_{i=1}^{m} E(t_i) = \sum_{i=1}^{m} \frac{m}{(m-(i-1))} = m \times H_m \tag{7.5}$$

Here, $E(t_i)$ is expected time of visiting a new area after i - 1 areas have been visited. H is a harmonic number.

For example, consider the situation in which we do not want to change areas before we visit approximately 80% of the areas. Then, the expected time T' to visit 80% of the areas is evaluated as follows. We can allocate $Threshold$ as follows.

$$k = \lfloor 0.2 \times m \rfloor \tag{7.6}$$

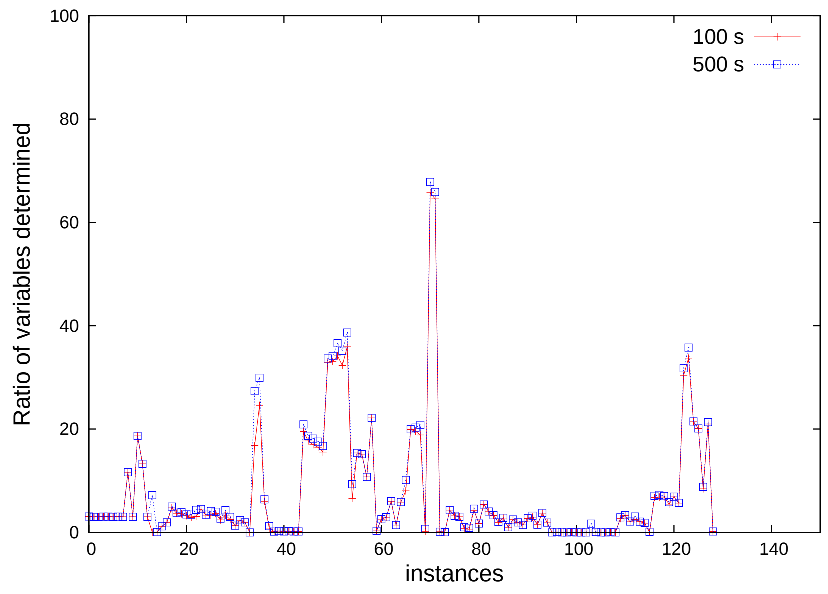
$$E(T') = \sum_{i=1}^{m-k} E(t_i) = m \times (H_m - H_k) \tag{7.7}$$

Next, we set $Threshold$ as follows.

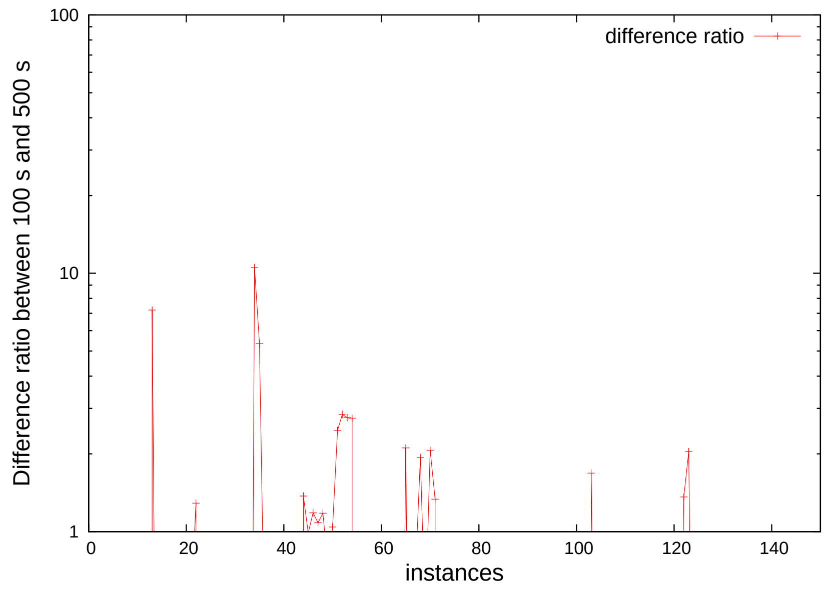
$$Threshold = E(T')/n \tag{7.8}$$

Here, we do not change areas if the density value is less than $Threshold$ because we only update a map every n conflicts; thus, it would be reasonable to divide $E(T')$ by n. There exists a problem for $E(T')$ calculation, because $H(m) \simeq m \times \ln 2$, thus very large integer would be required. Therefore, $E(T')$ increases exponentially in proportion to the size of search space. One possible way to avoid the curse of dimensionality is to pick several representative variables in each block for observation, and ignore others.

We must also consider variables that have unique values, i.e., a variable must always be *TRUE* or *FALSE*. In SaSS, we divide variables into blocks before the search; however, a large part of a block might become unique shortly after the search begins. We observed the ratios of unique variables after 100 and 500 s. Here, 300 benchmarks from the SAT-Race 2015 main track were used. Note that the instances solved within 500 s were removed. Figure 7.1.(a) compares the ratios obtained for 100 and 500 s. For several instances, the ratio was very high after only 100 s. We can easily resolve these problems by creating blocks after simplification processing. However, for several instances, the ratio increased



(a) 100 and 500 s



(b) Difference (ratio of 500 s - ratio of 100 s)

Fig. 7.1: Comparison of unique variable ratio between 100 and 500 s.

gradually, and the block size continued to change. Figure 7.1.(b) shows the differences of unique variables ratio between 100 s and 500 s. Y-values here are illustrated by logarithm scale, because most of them indicated zero, and most of non-zero values were also small. Note that the difference of the unique variables ratio is not negligible in several instances. Thus, we consider merging blocks dynamically. SaSS changes the cells by selecting a block and changing the status of that block. If a large part of a block is occupied by unique variables, the block size decreases. In this case, this block can be merged with another block. When we merge blocks, the aforementioned *Threshold* for evaluating the density must also be changed dynamically.

7.2 Resemble structures

Proposals and ideas in this section are based on several assumptions. In current status, implementation of them might boost the performance of SAT solvers, but there remains a correctness problem that a satisfiable instance is recognized by an unsatisfiable instance. We may avoid this problem, but we are not sure this is more efficient than the current modern SAT solvers.

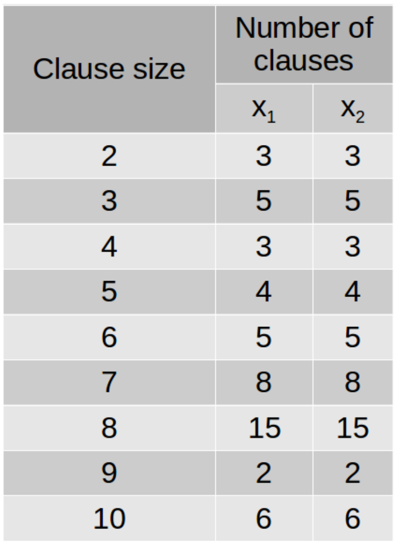
We assumes that application problems might have repeated structures in them. When a SAT solver find a solution of an application problem, it learns only one clause from each conflict. If we can extract repeated structures in a problem, we might add several clauses from a learned clause.

As a preliminary test, we observed an instance 002-80-8.cnf for binary connections, as shown in Equations 7.9 and 7.10. Variables x and y are mutually connected, because an assignment of one forces that of the other.

$$x \leftrightarrow y \text{ or } \neg x \leftrightarrow \neg y \iff (x \vee \neg y) \wedge (\neg x \vee y) \tag{7.9}$$

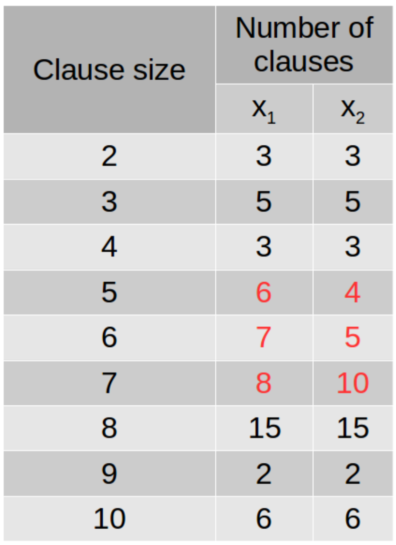
$$\neg x \leftrightarrow y \text{ or } x \leftrightarrow \neg y \iff (x \vee y) \wedge (\neg x \vee \neg y) \tag{7.10}$$

The 002-80-8.cnf instance contains 13408 variables and 478484 clauses. We observed 478484 clauses and counted the number of pairs having a relation of either Equation 7.9 or 7.10. There were 41107 and 41214 relations for Equations 7.9 and 7.10, respectively.



Clause size	Number of clauses	
	x_1	x_2
2	3	3
3	5	5
4	3	3
5	4	4
6	5	5
7	8	8
8	15	15
9	2	2
10	6	6

(a) Candidate pair



Clause size	Number of clauses	
	x_1	x_2
2	3	3
3	5	5
4	3	3
5	6	4
6	7	5
7	8	10
8	15	15
9	2	2
10	6	6

(b) Excluded pair

Fig. 7.2: Example of assumption for candidate pair for resemble structures.

For example for Equation 7.9, we counted a relation between x and y, if 478484 clauses include both clauses $(x \vee \neg y)$ and $(\neg x \vee y)$. Therefore, many instance might have

Let us consider x and y are connected to other variables with exactly the same ways. Then, a solver might add a clause $(\neg x \vee y)$ when it learns a clause $(x \vee \neg y)$. However, it seems to look time-consuming process for finding pairs of x and y. We can attempt to find x and y with the approximations. We explain the concept of our approach with Figure 7.2. As finding exact pairs is difficult, we can find candidate pairs first. To become a candidate pair, two variables should have the same clause size distributions. Figure 7.2 shows the clause size distributions for x_1 and x_2. Each number in rows 2 or 3 indicates the number of clauses x_1 or x_2 is included, respectively. As size distributions are the same in Figure 7.2.(a), x_1 and x_2 can be a candidate pair. However in Figure 7.2.(b), a pair of x_1 and x_2 can be excluded because of their differences. As size distributions are not sufficient, we additionally consider the distance on a variable graph between variables in a pair. Consider we have candidate pairs of (x_1, x_2) and (x_3, x_4), and their distance are $dist(x_1, x_3) = 4$, $dist(x_1, x_4) = 12$, $dist(x_2, x_3) = 4$, and $dist(x_2, x_4) = 7$. Then, $dist(x_1, x_3) = dist(x_2, x_3)$, and we might add a clause $(x_2 \vee x_3)$ if $(x_1 \vee x_3)$ is learned. On the other hand, we cannot add a clause $(x_2 \vee x_3)$ from a clause (x_2, x_4).

As we abovementioned, adding wrong clauses might turn a satisfiable instance into an unsatisfiable instance. Therefore, the more conditions we check the validity of a candidate pair, the more wrong wrong pairs are eliminated, but more conditions require more of time. Conditions mentioned above were not sufficient in our experiments. We designed several conditions for the calculation of resemblance between two variables stated below.

$$resemblance(x,y) = \prod_{i=1}^{4} cdn_i(x,y) \tag{7.11}$$

$$numCl(x,k)\text{: Number of clauses with a length } k \text{ including a variable } x \tag{7.12}$$

$$nonZeroCl(x,k) = \begin{cases} 1 & \text{if } numCl(x,k)! = 0 \\ 0 & \text{otherwise} \end{cases} \tag{7.13}$$

$$b(x,y,k) = \begin{cases} 1 & \text{if } numCl(x,k) = numCl(y,k) \\ 0 & \text{otherwise} \end{cases} \tag{7.14}$$

$$cdn_1(x,y) = \prod_{i=1}^{10} b(x,y,i) \tag{7.15}$$

$$cdn_2(x,y) = \begin{cases} 1 & \text{if } \sum_{i=1}^{10} numCl(x,k) \geq 100 \;\&\&\; \sum_{i=1}^{10} numCl(y,k) \geq 100 \\ 0 & \text{otherwise} \end{cases} \tag{7.16}$$

$$cdn_3(x,y) = \begin{cases} 1 & \text{if } \sum_{i=1}^{10} nonZeroCl(x,k) \geq 3 \;\&\&\; \sum_{i=1}^{10} nonZeroCl(y,k) \geq 3 \\ 0 & \text{otherwise} \end{cases} \tag{7.17}$$

$$surroundings(x,k)\text{: Number of variables within a distance } k \text{ in a variable graph} \tag{7.18}$$

$$resembleSurroundings(x,y,k) = \begin{cases} 1 & \text{if } surroundings(x,k) = surroundings(y,k) \\ 0 & \text{otherwise} \end{cases} \tag{7.19}$$

$$cdn_4(x,y) = \prod_{i=1}^{10} resembleSurroundings(x,y,i) \tag{7.20}$$

Resemblances of pairs can be obtained as a preprocessing, because we are only interested in resemblances in the orginal SAT formula. Resemblance between x and y has one when all four conditions are satisfied, as shown in Equation 7.11. Condition 1 compares clause size distributions between x and y, as shown in Equation 7.15. Condition 2 is designed to elimiate pairs when they are connected only a few clauses, as shown in Equation 7.16. Condition 3 is designed to elimiate pairs when variables in a pair are only connected to several kinds of clause sizes, as shown in Equation 7.17. Condition 4 compares sizes of variable graphs of x and y with a diameter limitation when x and y are a center of their graphs, respectively. In summary, we designed four conditions for an attempt to reduce the comparison time and increase the accurary of resemblances. We implemented these equations and performed them to several instances. When a clause $(x \vee y)$ is learned, a solver adds clauses $(x \vee z)$ and $(y \vee w)$, where $resemblance(x, z) = 1$ and $resemblance(y, w) = 1$, respectively. In our experiments, a solver with these conditions does not changed a satisfiable instance into an unsatisfiable instance. However, its efficiency was not good, because only $0.05n$ clauses are added by our method, when n binary clauses are found. We might obtain better performances if we improve conditions for the resemblance. Further design would be required, and further verification would also required for applying resemble structures in a SAT solver.

7.3 Reshuffling

One of the most influential techniques for SAT solvers is a restart technique. Modern SAT solvers restarts frequently for diversifying searches. However, a solver falls into deserts from time to time. Additional diversification method would be required to escape from deserts. Therefore, we designed *recursive reshuffling* option which performs *ISoV* recursively stated in a Chapter 3. Applying *recursive reshuffling* to a sequential solver might not a good idea, when we consider a balance between diversification and intensification. This option might strengthen search diversificaion a lot. In our assumption, applying only to several workers in a parallel SAT solver might be a good idea for search diversification. Details for *recursive reshuffling* are as follows.

- *MTL*: Maximum time limit for recursive search

- *EN*: Expected number of recursive searches in time MTL
- *TM*: Time margin to perform reshuffling process
- *tr*[*i*]: Time limit for recursive reshuffling (i) times

$$tr[i] = \frac{MTL - i \times ET}{i+1}$$

- t_i: Time limit of i-th worker for reshuffling

$$t_i = tr[i\%MN]$$

We implemented this on ParaGluminisat and tested using 12 workers with MTL to 3600, *MN* to 30, and ET to 100. Therefore in our experiments, $tr = \{3600$, 1750, 1133, 825, 640, 516, 428, 362, 311, $270\}$ corresponding time limit for recursive search for each worker respectively. Unfortunately, experiments with *recursive reshuffling* option showed undesirable results, because this option is activated in all workers except for 0-worker. However, we presume that this option might lead to better results, if the number of workers is high and only a few workers active this. Extra experiments are required to confirm this hypothesis.

Chapter 8

Conclusions and future directions

In this chapter, we summarize our research in terms of results and methods. We also discuss several future directions for improving our proposals.

8.1 Analysis of results

We proposed some approaches for adjusting the diversification and intensification to improve the performance of SAT solvers.

First, we proposed a *tie-breaking* method that is applicable to many existing branching heuristics. We generated TBVSIDS and TBCHB from VSIDS and CHB, respectively. These were evaluated in terms of 900 application benchmarks from SAT Competitions. We succeeded in improving branching heuristics by applying our *tie-breaking* method. The CHB is a recently proposed branching heuristic, but VSIDS is widely being used in many modern SAT solvers. Therefore, the improvement of VSIDS is significant for a wide range of SAT solvers.

Secondly, we discussed a hybrid branching heuristic as an algorithm selection in SAT research. We proposed a large framework for the integration of SAT solvers. As a first step in achieving this, we constructed a hybrid branching heuristic model. As SAT solvers are diverse, constructing a model targeted for a single solver is not desirable. Therefore, we evaluated our model using different solvers. We constructed a model using random forest, using 23 features. Ten of the 23 features were infeasible to extract for large instances. Therefore, we used 13 features first, and applied random sampling for extracting 23 features. As the performance of random sampling was adequate, we were able to find a better model using 23 features than using 13 features.

Finally, we proposed the AHM, which is applicable to adjusting diversification and intensification. To achieve the AHM in SAT solvers, we suggested the *PSSI* index by mapping the current assignment into an integer. We proposed the SaSS heuristic as an application for utilizing the AHM. The SaSS heuristic was evaluated using the 300 benchmarks from SAT-Race 2015 and showed improved results.

8.2 Analysis of methods

We analyze our proposals in terms of their methods.

First, *tie-breaking* in a branching heuristic is an attempt to select a more valuable variable from ties. We noticed that ties occur frequently in VSIDS, a widely used branching heuristic, and considered a smart *tie-breaking* method to improve the search efficiency. Since we are attempting to pick a better one from good ones, it would be desirable to extract it rapidly. We also considered the interplay between a branching heuristic and a clause learning scheme for efficient search intensification.

Second, we primarily showed the results of a hybrid branching heuristic. We applied random forest, because many features can be extracted from SAT instances, and distances between instances in a feature space appear to be irrelevant. Previous studies of algorithm selection for SAT solvers gathered a number of solvers and developed a model for assigning an adequate solver for each instance. In contrast to these multi-solver approaches, we proposed a single-solver with hybrid models. As long as a solver maintains a single-solver form, it can be utilized for ongoing research as a base solver.

Finally, we evaluated our AHM in a parallel SAT solver. The construction of an AHM for a SAT solver is an attempt to summarize the nogood signals and share among workers. Several factors are considered for efficiency in the AHM design. First, we use a single history map, rather than pairwise comparisons among pairs of workers. Second, the size of a single history map is fixed, and it uses only a small amount of memory. Third, we integrated information only after a conflict to gather nogoods. Therefore, the AHM is applicable for massively parallel environments with low memory requirements and low computational times.

8.3 Suggestions for future directions

We address several ideas for future work that can improve or extend applicable areas.

First, we discuss about *tie-breaking*. We assessed the quality of each clause through an inverse of its LBD. However, we can also consider its variants such as the inverse of LBD^2 or the inverse of 2^{LBD}. During our experiments, we noticed that the strengthening of the interplay between a branching heuristic and a clause learning scheme is very important. For example in the modern VSIDS, variables related to resolutions, i.e., conflict analysis, are incremented by one. We can also consider adjusting their scores based on their distances from a conflict. A branching heuristic might be required to utilize learned clauses efficiently. Therefore, additional ideas with consideration of interplays among components in a SAT solver might improve its performance. We evaluated *tie-breaking* using VSIDS and CHB. Application of *tie-breaking* into a LRB, a recently proposed new branching heuristic, might generate a more efficient branching heuristic.

Second, we addressed and evaluated a hybrid branching heuristic model as a component in our hybrid concept. Our model is evaluated using two solvers. A model might become more general if we consider a third solver for evaluation. Random sampling is applied for avoiding out-of-time exceptions for infeasible features. Additional analysis or design would be required, because random sampling worked well in our experiments, but this might have the potential of degrading a model when new instances are added. A dynamic method such as stopping the extraction process when a feature value is stabilized might be useful. Currently, we evaluated only a hybrid branching heuristic model. We might construct a hybrid restart model or a hybrid clause learning model, and attempt to integrate these models in a single solver.

Third, we introduced the AHM. An application of the SaSS heuristic using AHM was experimentally evaluated, and our intuition tells us that the SaSS heuristic worked well because of the search diversification. We suggested additional ideas for improving SaSS, including biased random work and density evaluation. Further ideas might be required, but their implementation and evaluation can be future work. The SaSS is one possible scenario for utilizing the AHM. We might evaluate the potential of the AHM using other scenarios, such as dividing workers into several groups, intensifying workers in the same group, and diversifying among groups. Constructing an AHM in another research category, such as an AHM for the MAX-SAT solver, can be considered as future work.

Publications and Research Activities

Conferences (With reviews)

(1) S. Moon and M. Inaba, Dynamic Strategy to Diversify Search Using a History Map in Parallel Solving. International Conference on Learning and Intelligent Optimization 10 (LION 10), pp. 260–266. Ischia Island, Italy, 2016.

(2) S. Moon and M. Inaba, Boost SAT Solver with Hybrid Branching Heuristic, The 10th Annual Symposium on Combinatorial Search (SoCS 2017) pp 56–63. Pittsburgh, USA, 2017.

Workshops (With reviews)

(3) S. Moon and M. Inaba, Approximate History Map for Massively Parallel Environments. Pragmatics of SAT 2016 (PoS-16), Bordeaux, France, 2016.

(4) S. Moon and M. Inaba, Evaluation of New Branching Heuristic for Tie-breaking. The 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016) doctral program, Toulouse, France, 2016.

Domestic workshops

(5) S. Moon and M. Inaba, Simple Method to Diversify Search Space in SAT Problems. The 29th Annual Conference of the Japan Society of Artificial Intelligence (JSAI 2015), Hakodate, 2015.

(6) S. Moon and M. Inaba, Boost SAT solver with hybrid branching heuristic. The 31th Annual Conference of the Japan Society of Artificial Intelligence (JSAI 2017), Nagoya, 2017.

Awards

(7) TC_Glucose, Best Crafted Benchmark Solver in the Main Track, SAT Competition, 2016.

(8) TB_Glucose, Silver medal in the Agile Track, SAT Competition, 2016.

(9) CHBR_Glucose, Bronze medal in the Agile Track, SAT Competition, 2016.

(10) Student Challenge Winner, SAT Conference, 2016.

References

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, ACM, 1971.

[2] W. I. Gasarch, "Guest column: The second p =?np poll," *SIGACT News*, vol. 43, pp. 53–77, June 2012.

[3] E. Friedgut, J. Bourgain, *et al.*, "Sharp thresholds of graph properties, and the k-sat problem," *Journal of the American mathematical Society*, vol. 12, no. 4, pp. 1017–1054, 1999.

[4] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham, "Understanding random sat: Beyond the clauses-to-variables ratio," in *International Conference on Principles and Practice of Constraint Programming*, pp. 438–452, Springer, 2004.

[5] H. A. Kautz, B. Selman, *et al.*, "Planning as satisfiability.," in *ECAI*, vol. 92, pp. 359–363, Citeseer, 1992.

[6] D. Jackson and M. Vaziri, "Finding bugs with a constraint solver," in *ACM SIGSOFT Software Engineering Notes*, vol. 25, pp. 14–25, ACM, 2000.

[7] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal methods in system design*, vol. 19, no. 1, pp. 7–34, 2001.

[8] V. Paruthi and A. Kuehlmann, "Equivalence checking combining a structural sat-solver, bdds, and simulation," in *Computer Design, 2000. Proceedings. 2000 International Conference on*, pp. 459–464, IEEE, 2000.

[9] Z. Newsham, W. Lindsay, V. Ganesh, J. H. Liang, S. Fischmeister, and K. Czarnecki, "Satgraf: Visualizing the evolution of sat formula structure in solvers," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 62–70, Springer, 2015.

[10] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding

of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[11] `http://www.satcompetition.org/`.

[12] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[13] A. Biere, M. Heule, H. van Maaren, and T. Walsh, "Conflict-driven clause learning sat solvers," *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pp. 131–153, 2009.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535, ACM, 2001.

[15] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Exponential recency weighted average branching heuristic for sat solvers.," in *AAAI*, pp. 3434–3440, 2016.

[16] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla-07: the design and analysis of an algorithm portfolio for sat," in *International Conference on Principles and Practice of Constraint Programming*, pp. 712–727, Springer, 2007.

[17] A. Loreggia, Y. Malitsky, H. Samulowitz, and V. A. Saraswat, "Deep learning for algorithm portfolios.," in *AAAI*, pp. 1280–1286, 2016.

[18] A.-H. A. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng, "Evaluating the impact of memory system performance on software prefetching and locality optimizations," in *Proceedings of the 15th international conference on Supercomputing*, pp. 486–500, ACM, 2001.

[19] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–24, 2011.

[20] C. Sinz, "Visualizing the internal structure of sat instances (preliminary report)," in *SAT*, Citeseer, 2004.

[21] C. Ansótegui, J. Giráldez-Cru, and J. Levy, "The community structure of sat formulas," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 410–423, Springer, 2012.

[22] S. Arora and B. Barak, *Computational complexity: a modern approach.* Cambridge University Press, 2009.

[23] G. Tseitin, "On the complexity ofderivation in propositional calculus," *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[24] B. Selman, H. J. Levesque, D. G. Mitchell, *et al.*, "A new method for solving hard satisfiability problems.," in *AAAI*, vol. 92, pp. 440–446, 1992.

[25] B. Mazure, L. Sais, and É. Grégoire, "Tabu search for sat," in *AAAI/IAAI*, pp. 281–285, 1997.

[26] H. Kautz and B. Selman, "Pushing the envelope: Planning, propositional logic, and stochastic search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 1194–1201, 1996.

[27] T. Larrabee, "Test pattern generation using boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4–15, 1992.

[28] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM (JACM)*, vol. 7, no. 3, pp. 201–215, 1960.

[29] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *International conference on theory and applications of satisfiability testing*, pp. 61–75, Springer, 2005.

[30] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *International conference on tools and algorithms for the construction and analysis of systems*, pp. 129–144, Springer, 2010.

[31] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," *Automated Reasoning*, pp. 355–370, 2012.

[32] `http://baldur.iti.kit.edu/sat-competition-2016/`.

[33] G. Chu, A. Harwood, and P. J. Stuckey, "Cache conscious data structures for boolean satisfiability solvers," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 99–120, 2008.

[34] J. P. M. Silva and K. A. Sakallah, "Grasp-a new search algorithm for satisfiability," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pp. 220–227, IEEE Computer Society, 1997.

[35] H. Zhang and M. E. Stickel, "An efficient algorithm for unit propagation," in *In Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH96), Fort Lauderdale Florida USA*, pp. 166–169, 1996.

[36] H. Zhang, "Sato: An efficient prepositional prover," in *International Conference on Automated Deduction*, pp. 272–275, Springer, 1997.

[37] I. Lynce and J. Marques-Silva, "Efficient data structures for backtrack search sat solvers," *Annals of Mathematics and Artificial Intelligence*, vol. 43, no. 1-4, pp. 137–152, 2005.

[38] L. Zhang and S. Malik, "Cache performance of sat solvers: A case study for efficient implementation of algorithms," in *International conference on theory and applications of satisfiability testing*, pp. 287–298, Springer, 2003.

[39] J. Marques-Silva, "Algebraic simplification techniques for propositional satisfiability," in *International Conference on Principles and Practice of Constraint Programming*, pp. 537–542, Springer, 2000.

[40] B. Aspvall, M. F. Plass, and R. E. Tarjan, "A linear-time algorithm for testing the truth of certain quantified boolean formulas," *Information Processing Letters*, vol. 8, no. 3, pp. 121–123, 1979.

[41] R. J. Bayardo Jr and R. Schrag, "Using csp look-back techniques to solve real-world sat instances," in *AAAI/IAAI*, pp. 203–208, 1997.

[42] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pp. 279–285, IEEE Press, 2001.

[43] Y. Gurevich and D. G. Mitchell, "A sat solver primer," 2005.

[44] L. Simon and G. Audemard, "Predicting Learnt Clauses Quality in Modern SAT Solver," in *Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, (Pasadena, United States), July 2009.

[45] M. Buro and H. K. Büning, *Report on a SAT competition.* Fachbereich Math.-Informatik, Univ. Gesamthochschule, 1992.

[46] J. Marques-Silva, "The impact of branching heuristics in propositional satisfiability algorithms," in *Portuguese Conference on Artificial Intelligence*, pp. 62–74, Springer, 1999.

[47] R. G. Jeroslow and J. Wang, "Solving propositional satisfiability problems," *Annals of mathematics and Artificial Intelligence*, vol. 1, no. 1-4, pp. 167–187, 1990.

[48] J. N. Hooker and V. Vinay, "Branching rules for satisfiability," *Journal of Automated Reasoning*, vol. 15, no. 3, pp. 359–383, 1995.

[49] J. W. Freeman, *Improvements to propositional satisfiability search algorithms.* PhD thesis, University of Pennsylvania, 1995.

[50] A. Van Gelder and Y. Tsuji, *Satisfiability testing with more reasoning and less guessing.* Computer Research Laboratory [University of California, Santa Cruz], 1995.

[51] C. M. Li, "A constraint-based approach to narrow search trees for satisfiability," *Information processing letters*, vol. 71, no. 2, pp. 75–80, 1999.

[52] N. Eén and N. Sörensson, "An extensible sat-solver," in *International conference on theory and applications of satisfiability testing*, pp. 502–518, Springer, 2003.

[53] A. Biere, "Adaptive restart strategies for conflict driven sat solvers," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 28–33, Springer, 2008.

[54] A. Biere and A. Fröhlich, "Evaluating cdcl variable scoring schemes," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 405–422, Springer, 2015.

[55] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust sat-solver," *Discrete Applied Mathematics*, vol. 155, no. 12, pp. 1549–1561, 2007.

[56] N. Dershowitz, Z. Hanna, and A. Nadel, "A clause-based heuristic for sat solvers," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 46–60, Springer, 2005.

[57] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for sat solvers," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 123–140, Springer, 2016.

[58] N. Sorensson, "Minisat 2.2 and minisat++ 1.1," in *A short description in SAT Race 2010*, 2010.

[59] Y. Hamadi, S. Jabbour, and J. Sais, *Control-Based Clause Sharing in Parallel SAT Solving*, pp. 245–267. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[60] K. Pipatsrisawat and A. Darwiche, *A Lightweight Component Caching Scheme for Satisfiability Solvers*, pp. 294–299. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[61] H. Zhang, M. P. Bonacina, and J. Hsiang, "Psato: a distributed propositional prover and its application to quasigroup problems," *Journal of Symbolic Computation*, vol. 21, no. 4, pp. 543–560, 1996.

[62] W. Chrabakh and R. Wolski, "The gridsat portal: a grid web-based portal for solving satisfiability problems using the national cyberinfrastructure," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 6, pp. 795–808, 2007.

[63] L. Gil, P. Flores, and L. M. Silveira, "Pmsat: a parallel version of minisat," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 71–98, 2008.

[64] B. A. Huberman, R. M. Lukose, and T. Hogg, "An economics approach to hard computational problems," *Science*, vol. 275, no. 5296, pp. 51–54, 1997.

[65] C. P. Gomes and B. Selman, "Algorithm portfolio design: Theory vs. practice," in *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, pp. 190–197, Morgan Kaufmann Publishers Inc., 1997.

[66] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2008.

[67] Y. Hamadi, S. Jabbour, and J. Sais, "Control-based clause sharing in parallel sat solving," in *Autonomous Search*, pp. 245–267, Springer, 2011.

[68] C. BahMing and J. Raj, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN Systems*, vol. 1, p. 1414, 1939.

[69] G. Audemard and L. Simon, "Lazy clause exchange policy for parallel sat solvers," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 197–205, Springer, 2014.

[70] G. Audemard, B. Hoessen, S. Jabbour, J.-M. Lagniez, and C. Piette, "Revisiting clause exchange in parallel sat solving," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 200–213, Springer, 2012.

[71] G. Audemard, J.-M. Lagniez, B. Mazure, and L. Saïs, "On freezing and reactivating learnt clauses," in *International Conference on Theory and Applications of*

Satisfiability Testing, pp. 188–200, Springer, 2011.

[72] A. Irfan, D. Lanti, and N. Manthey, “Modern cooperative parallel sat solving,” 2013.

[73] N. Totla and A. Devarakonda, “Massive parallelization of sat solvers,”

[74] G. Audemard, J.-M. Lagniez, N. Szczepanski, and S. Tabary, “An adaptive parallel sat solver,” in *International Conference on Principles and Practice of Constraint Programming*, pp. 30–48, Springer, 2016.

[75] G. Audemard, B. Hoessen, S. Jabbour, and C. Piette, “An effective distributed d&c approach for the satisfiability problem,” in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pp. 183–187, IEEE, 2014.

[76] A. Semenov and O. Zaikin, “Using monte carlo method for searching partitionings of hard variants of boolean satisfiability problem,” in *International Conference on Parallel Computing Technologies*, pp. 222–230, Springer, 2015.

[77] T. Balyo, P. Sanders, and C. Sinz, “Hordesat: a massively parallel portfolio sat solver,” in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 156–172, Springer, 2015.

[78] J. M. Crawford and A. B. Baker, “Experimental results on the application of satisfiability algorithms to scheduling problems,” in *AAAI*, vol. 2, pp. 1092–1097, 1994.

[79] C. P. Gomes, B. Selman, and N. Crato, “Heavy-tailed distributions in combinatorial search,” in *International Conference on Principles and Practice of Constraint Programming*, pp. 121–135, Springer, 1997.

[80] C. P. Gomes, B. Selman, H. Kautz, *et al.*, “Boosting combinatorial search through randomization,” *AAAI/IAAI*, vol. 98, pp. 431–437, 1998.

[81] A. Biere, “Picosat essentials,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, pp. 75–97, 2008.

[82] M. Luby, A. Sinclair, and D. Zuckerman, “Optimal speedup of las vegas algorithms,” *Information Processing Letters*, vol. 47, no. 4, pp. 173–180, 1993.

[83] T. Walsh *et al.*, “Search in a small world,” in *IJCAI*, vol. 99, pp. 1172–1177, 1999.

[84] C. Sinz and M. Iser, “Problem-sensitive restart heuristics for the dpll procedure,” in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 356–362, Springer, 2009.

[85] K. Pipatsrisawat and A. Darwiche, "Width-based restart policies for clause-learning satisfiability solvers," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 341–355, Springer, 2009.

[86] V. Ryvchin and O. Strichman, "Local restarts," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 271–276, Springer, 2008.

[87] G. Audemard and L. Simon, "Refining restarts strategies for sat and unsat," in *Principles and Practice of Constraint Programming*, pp. 118–126, Springer, 2012.

[88] L. Guo, Y. Hamadi, S. Jabbour, and L. Sais, "Diversification and intensification in parallel sat solving," in *International conference on principles and practice of constraint programming*, pp. 252–265, Springer, 2010.

[89] L. Guo and J. M. Lagniez, "Dynamic polarity adjustment in a parallel sat solver," in *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pp. 67–73, Nov 2011.

[90] G. Audemard and L. Simon, *Experimenting with Small Changes in Conflict-Driven Clause Learning Algorithms*, pp. 630–634. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[91] L. Simon, D. Le Berre, and E. A. Hirsch, "The sat2002 competition," *Annals of Mathematics and Artificial Intelligence*, vol. 43, no. 1, pp. 307–342, 2005.

[92] M. Nikolić, *Statistical Methodology for Comparison of SAT Solvers*, pp. 209–222. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[93] H. Toutenburg, "Fisher, r. a., and f. yates: Statistical tables for biological, agricultural and medical research. 6th ed. oliver & boyd, edinburgh and london 1963. x, 146 p. preis 42 s net," *Biometrische Zeitschrift*, vol. 13, no. 4, pp. 285–285, 1971.

[94] C. Oh, "Minisat hack 999ed, minisat hack 1430ed and swdia5by," *SAT Competition*, 2014.

[95] H. NABESHIMA, K. IWANUMA, and K. INOUE, "Glueminisat 2.2.5: A fast sat solver with an aggressive acquiring strategy of glue clauses," *Computer Software*, vol. 29, no. 4, pp. 146–160, 2012.

[96] E. Huang and R. E. Korf, "New improvements in optimal rectangle packing.," in *IJCAI*, pp. 511–516, 2009.

[97] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: an auto-

matic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, 2009.

[98] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki, "Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers," in *Haifa Verification Conference*, pp. 225–241, Springer, 2015.

[99] B. Tomoiagă, M. Chindriş, A. Sumper, A. Sudria-Andreu, and R. Villafafila-Robles, "Pareto optimal reconfiguration of power distribution systems using a genetic algorithm based on nsga-ii," *Energies*, vol. 6, no. 3, pp. 1439–1455, 2013.

[100] F. Hutter, H. Hoos, K. Leyton-Brown, K. Murphy, and S. Ramage, "Smac: Sequential model-based algorithm configuration," 2011.

[101] J. R. Rice, "The algorithm selection problem," *Advances in computers*, vol. 15, pp. 65–118, 1976.

[102] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Algorithm selection and scheduling," in *International Conference on Principles and Practice of Constraint Programming*, pp. 454–469, Springer, 2011.

[103] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, "Evaluating component solver contributions to portfolio-based algorithm selectors," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 228–241, Springer, 2012.

[104] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Algorithm portfolios based on cost-sensitive hierarchical clustering.," in *IJCAI*, 2013.

[105] M. G. Lagoudakis and M. L. Littman, "Learning to select branching rules in the dpll procedure for satisfiability," *Electronic Notes in Discrete Mathematics*, vol. 9, pp. 344–359, 2001.

[106] C. Oh, "Between sat and unsat: the fundamental difference in cdcl sat," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 307–323, Springer, 2015.

[107] J. Chen, "Improving abcdsat by at-least-one recently used clause management strategy," *CoRR*, vol. abs/1605.01622, 2016.

[108] `http://http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud/`.

[109] J. Bader and E. Zitzler, "Hype: An algorithm for fast hypervolume-based many-

objective optimization," *Evolutionary computation*, vol. 19, no. 1, pp. 45–76, 2011.

[110] P. Kilby, J. Slaney, S. Thiébaux, T. Walsh, *et al.*, "Backbones and backdoors in satisfiability," in *AAAI*, vol. 5, pp. 1368–1373, 2005.

[111] M. Ferrante and M. Saltalamacchia, "The coupon collectors problem," *Materials matemàtics*, pp. 0001–35, 2014.