International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2022, 12(1): 131–151, doi: 10.21655/ijsi.1673-7288.00278
©2022 by Institute of Software, Chinese Academy of Sciences. All rights reserved.

Research Article

Branching Strategy Selection Approach Based on Vivification Ratio

Mao Luo[1], Chumin Li[1,2,4], Xinyun Wu[3], Shuolin Li[4], Zhipeng Lü[1]

[1] (School of Computer Science, Huazhong University of Science and Technology, Wuhan 430074, China)
[2] (MIS, University of Picardie Jules Verne, Amiens CS 52501, France)
[3] (School of Computer Science, Hubei University of Technology, Wuhan 430068, China)
[4] (Aix Marseille Univ., University of Toulon, CNRS, LIS, Marseille CS 60584, France)
Corresponding author: Chumin Li, chu-min.li@u-picardie.fr

Abstract The two most effective branching strategies LRB and VSIDS perform differently on different types of instances. Generally, LRB is more effective on crafted instances, while VSIDS is more effective on application ones. However, distinguishing the types of instances is difficult. To overcome this drawback, we propose a branching strategy selection approach based on the vivification ratio. This approach uses the LRB branching strategy more to solve the instances with a very low vivification ratio. We tested the instances from the main track of SAT competitions in recent years. The results show that the proposed approach is robust and it significantly increases the number of solved instances. It is worth mentioning that, with the help of our approach, the solver Maple_CM can solve additional 16 instances for the benchmark from the 2020 SAT competition.

Keywords satisfiability; conflict-driven clause learning; branching heuristics; clause vivification

Citation Luo M, Li CM, Wu XY, Li SL, Lü ZP. Branching strategy selection approach based on vivification ratio. *International Journal of Software and Informatics*, 2022, 12(1): 131–151. http://www.ijsi.org/1673-7288/278.htm

1 Introduction

The SAT problem consists of finding an assignment to all the variables in a propositional logic formula ϕ in Conjunctive Normal Form (CNF), to satisfy all clauses in ϕ. SAT is the first problem proven to be NP-complete. Thus, many NP problems can be solved by translating them into a SAT equivalent or by considering SAT as a core part of the solving process. SAT is widely used in various areas, especially in the automation of circuits design, including Equivalence Checking[1], Formal Verification[2], Automatic Test Pattern Generation[3, 4], Model Checking, Logic Synthesis[5], software and hardware checking[6–8], planning[9, 10], and scheduling[11]. SAT also affects the research of many related decision and optimization problems[4, 12–15].

For SAT solving, there are two mainstream types of algorithms: complete algorithms and incomplete algorithms. Complete algorithms could prove unsatisfiability, while incomplete algorithms could not prove unsatisfiability but could solve some certain types of satisfiable

Funding items: French Agence Nationale de la Recherche (ANR-19-CHIA-0013-01)
Received 2021-11-21; Revised 2021-12-07; Accepted 2021-12-08; IJSI published online 2022-03-28

instances very efficiently. For complete algorithms, Conflict-Driven Clause Learning (CDCL) solvers and look-ahead solvers achieve success. CDCL solvers could solve industrial application SAT instances very fast; look-ahead solvers are able to solve unsatisfiable random SAT instances efficiently[16–18]; recently, the combination of CDCL solvers and look-ahead solvers made a breakthrough in automated theory proving[19]. For incomplete algorithms, Stochastic Local Search (SLS) solvers[20–24] and Survey Propagation (SP) solvers are very popular. SLS solvers show strong complementarity with CDCL solvers on solving a number of important application SAT instances, e.g., those instances encoded from station repacking[25, 26]; SP solvers show great effectiveness on solving very huge-sized random 3-SAT instances (e.g., with one million variables at the clause-to-variable ratio of 4.2)[27–32]. This paper focuses on modern CDCL SAT solvers which are very efficient for real applications.

The core techniques that guarantee the efficiency of the CDCL[33, 34] SAT solver include unit propagation, clause learning, branching strategy, clause simplification, management for the database of learnt clauses, restart, lazy data structure, etc. Clause simplification and branching strategy are among the techniques that gain increasing attention.

The methods based on clause simplification can be categorized into pre-processing and in-processing. The most effective pre-processing techniques include variants of Bounded Variable Elimination, Addition or Elimination of Redundant Clauses, Detection of Subsumed Clauses, and suitable combinations of them[35, 36]. They aim mostly at reducing the number of clauses, literals, and variables in the input formula. The most effective in-processing techniques[37] are Local and Recursive Clause Minimization[38, 39], On-the-fly Clause Subsumption[40, 41], and clause vivification[42, 43] where Local and Recursive Clause Minimization removes redundant literals from learnt clauses immediately after their creation, On-the-fly Clause Subsumption efficiently removes clauses subsumed by the resolvents derived during clause learning, and clause vivification periodically detects and removes redundant literals from clauses by unit propagation.

Early branching strategies are based on lookahead and choose the next decision variable by analyzing the clauses not yet satisfied during the searching process. The most classical branching strategies are MoMs[44, 45], Dynamic Literal Individual Sum Heuristic (DLIS)[46, 47] and UP[48]. These strategies do not lookback, i.e., they do not learn from what happened in the past to choose the next branching (decision) variable. More recently, lookback branching strategies have been introduced in CDCL SAT solvers, consisting in choosing the variables contributing most often to the recent conflicts. The contribution of the variables to the conflicts is collected during the clause learning driven by conflicts. VSIDS (Variable State Independent Decaying Sum)[34] and LRB (Learning Rate Branching)[49] are among the best lookback branching strategies. Different branching strategies may perform quite differently on different categories of instances, and no strategy outperforms others on all the instances. Many recent CDCL SAT solvers alternatively use VSIDS and LRB to combine their respective strength, independently of the instance to be solved. In these solvers, search is usually divided into pairs of phases, and in each pair of phases, LRB is used to select branching variables in one phase, and VSIDS is used in the other phase. Note that the two phases in the pair have the same length.

We believe that the use of branching strategies should be instance-dependent. In other words, when solving some families of instances, VSIDS should be used more often, while when solving some other families of instances, LRB should be used more often. Unfortunately, it is not easy to identify the families of instances for which a particular branching strategy should be used. In this paper, in order to improve the performance of CDCL SAT solvers, we propose to use the information gathered during clause vivification to decide which branching strategy to use more often. The approach is based on the observation that VSIDS appears to outperform LRB when clause vivification discovers many redundant literals in learnt clauses, and LRB appears to outperform VSIDS when clause vivification cannot discover many redundant literals in learnt clauses.

The paper is organized as follows: Section 2 gives some basic concepts about propositional satisfiability and CDCL SAT solvers. Section 3 presents some related work on branching strategies and combination of them. Section 4 gives a detailed analysis of the relationship between characteristics of different instances with branching strategy and vivification ratio firstly and then describes our branching strategy selection approach based on vivification ratio, as well as how it is implemented in general CDCL SAT solvers. Section 5 reports on the in-depth empirical investigation of the proposed branching strategy selection approach. Section 6 contains the concluding remarks.

2 Preliminaries

In propositional logic, a variable x may take the truth value 0 (false) or 1 (true). A literal l is a variable x or its negation $\neg x$, a clause C is a disjunction of literals, a CNF formula ϕ is a conjunction of clauses, and the size of a clause is the number of literals in it. An assignment of truth values to the propositional variables satisfies a literal x if it takes the value 1 and satisfies a literal $\neg x$ if it takes the value 0. An assignment satisfies a clause if it satisfies at least one of its literals and satisfies a CNF formula if it satisfies all of its clauses. The empty clause, denoted by $\square$, contains no literal and is unsatisfiable, i.e., it represents a conflict. A unit clause contains exactly one literal and is satisfied by assigning the appropriate truth value to the variable. An assignment for a CNF formula ϕ is complete if each variable in ϕ has been assigned a value; otherwise, it is said partial. The SAT problem for a CNF formula ϕ is to find an assignment to the variables satisfying all clauses of ϕ.

Modern state-of-the-art SAT solvers are based on the CDCL (Conflict Driven Clause Learning) scheme. The core of the scheme is to continuously generate conflicts and record conflicts through learning clauses. CDCL contains two phases, namely the search phase and the learning phase.

In the search phase, the most critical method is the Unit Propagation (UP) method, whose details are described as follows: If there exists a unit clause in ϕ, to satisfy this clause, the only literal l in it must be satisfied by assigning the appropriate truth value to the corresponding variable. The satisfaction of l implies the falsification of $\neg l$. Therefore, removing the literal $\neg l$ in other clauses and all clauses containing l does not change the satisfiability of the instance. After the removal, new unit clauses may appear and some clauses can become empty, i.e., no literal remains in the clauses. We can repeat this process to simplify the instance further until there is no unit clause or an empty clause is produced. This procedure is denoted as UP(ϕ) which returns a simplified formula that does not contain any unit clause, or a formula containing an empty clause.

If the UP procedure does not produce any conflict, the search will heuristically select a variable for assignment to make new UP possible. The selected variable is called decision variable and its referred literal is decision literal. The level of an assigned variable (by UP or by decision) is the number of decision variables so far. If all the variables are assigned and no conflict occurs after repeating the actions decision and UP, the formula ϕ is satisfiable. If an empty clause is produced during the unit propagation, a conflict occurs.

The learning phase starts after a conflict occurs. The process of conflict generation can be represented by an implication graph, like the one shown in Figure 1. Each vertex represents the satisfaction of a literal, marked as $l@dl$, where l denotes the literal, and dl denotes the decision level that the literal belongs to. The negation of the literals of incoming edges of a node $l@ld$

represents the reason (clause) why UP has set $l = 1$. For example, vertex x_3@3 is propagated by clause $x_1 \vee x_2 \vee x_3$ $(\neg x_1 \wedge \neg x_2 \rightarrow x_3)$. When the literals x_1 and x_2 are both assigned 0, x_3 must be assigned 1 to satisfy the clause. Conflict clauses are marked as incoming edges to $\Box$, and the level of conflict clauses is named conflict level. A decision literal is shown as a vertex marked in orange in the implication graph. It does not have any incoming edge. Note that each decision literal is the starting vertex of its decision level, and the decision literal of the conflict level is dashed. A Unique Implication Point (UIP) is a vertex that is in the conflict level and dominates all the paths to the conflict. Figure 1 contains two UIPs x_3@3 and $\neg x_1$@3. Among them, x_3@3 is the UIP closest to the conflict, called first UIP (FUIP), $\neg x_1$@3 is the UIP farthest from the conflict, called last UIP.

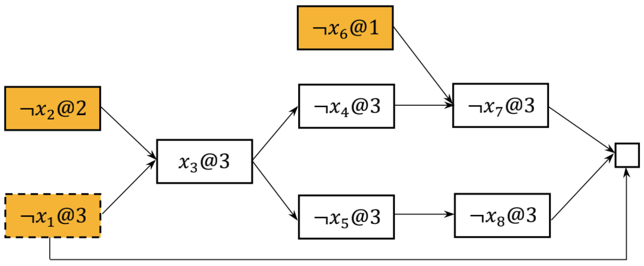

Figure 1 Example of implication graph

CDCL SAT solvers usually uses the first UIP-scheme for clause learning. Let *dl* be the conflict level. All literals of level *dl* on the path from FUIP to the conflict are called active literals. For example, in Figure 1, the conflict level is 3,and $\neg x_4$@3, $\neg x_5$@3, x_7@3, and $\neg x_8$@3 are active literals. In the FUIP scheme, the negation of each literal with level smaller than *dl* immediately preceding an active literal, as well as the negation of the FUIP, constructs the learnt clause. For example, the learnt clause of Figure 1 is $x_6 \vee \neg x_3$. The learnt clause stores the reason for the conflict and allows to avoid repeating the same assignment in the future. After adding the learnt clause, the CDCL solver backtracks to the second highest decision level and performs unit propagation. In Figure 1, it backtracks to level 1, making the learnt clause $x_6 \vee \neg x_3$ a unit clause $\neg x_3$, and unit clause propagation is performed. If there is a conflict in level 0, it proves that the instance is unsatisfiable.

The variables in the learnt clause C can be partitioned according to their level. The number of partitions in C is called the Literal Block Distance (LBD)[50]. For example, in Figure 1, the number of levels of the generated learnt clause x_6@1 $\vee \neg x_3$@3 is 2, so its LBD value is 2. A small LBD value usually indicates a clause of good quality.

Let $C = l_1 \vee \cdots \vee l_k$ be a clause in ϕ, l_i $(1 \leq i \leq k)$ be a literal in C, and $C \setminus l_i$ be C in which l_i is removed, and ϕ' be ϕ in which C is replaced by $C \setminus l_i$. If ϕ and ϕ' are equivalent, i.e., any solution of ϕ is also a solution of ϕ' and vice versa, then l_i is said to be a redundant literal in C. Clause vivification is an effective clause simplification technique to remove redundant literals in the clauses of ϕ, that can be described is as follows. If UP$(\phi \cup \{\neg l_1, \cdots, \neg l_{i-1}, \neg l_{i+1}, \cdots, \neg l_k\})$ results in an empty clause, then l_i is redundant in C. A clause vivification procedure executing UP$(\phi \cup \{\neg l_1, \neg l_2, \cdots, \neg l_i\})$ constructs an implication graph, where literals $\neg l_1, \neg l_2, \cdots$, and $\neg l_i$ can be considered as successive decision literals. For example, suppose that there is a clause $x_1 \vee x_2 \vee x_6 \vee x_9$, such that propagating successively $\neg x_1$, $\neg x_2$ and $\neg x_6$ forms the implication graph as shown in Figure 1, which implies a conflict. Since the literal x_9 does not participate in the conflict, the literal x_9 can be deleted from the clause. The Learnt Clause Minimization (LCM) approach as presented in Ref. [42] is a clause vivification technique consisting of eliminating redundant literals from learnt clauses.

3 Related Work

Branching is a core process of a CDCL SAT solver (see Refs. [34, 44–47, 49, 51–53]). A good branching strategy allows to quickly find the feasible solution of the problem or proves unsatisfiability. During the solving process of a CDCL SAT solver, the branching strategy analyzes the production of conflicts and guides the search process according to the information gathered during the conflict analysis.

The most successful branching strategy is Variable State Independent Decaying Sum (VSIDS)[34]. It computes the score of each variable as follows.

- The score of each variable is initialized to 0.
- In a learning phase, the score of each variable encountered in clause learning is increased by inc, where inc is a value initialized to 1.
- After each conflict, inc is increased to inc/d, where d is a constant usually fixed to 0.95, so that recent conflicts count more in the score of the variables.

The real applications usually have a clear community structure: The variables within the same community are constrained more strongly than those in different communities[54]. Keeping branching on the variables in the same community allows to better analyze the constraint relationship between them. Therefore, the performance of VSIDS on real application instances comes from its capacity to focus on the variables involved in recent conflicts within the same community, because these variables will have high score.

LRB[49] is a new branching strategy based on the Multi-Armed Bandit (MAB) framework in reinforcement learning. The score of each variable is the exponential recency-weighted average of the awards the variable received in the past, as defined below.

- The score $LRB(v)$ of each variable v is initialized to 0.
- During the search, a variable is frequently assigned a truth value and this assignment can be canceled later upon backtracking. For variable v, let t_1 be the number of conflicts produced when v is assigned a truth value, t_2 be the number of conflicts produced when this assignment is canceled, and k be the number of conflicts in which v is involved from t_1 conflicts to t_2 conflicts, then the reward r to v is defined to be $k/(t_2 - t_1)$ and the score $LRB(v)$ is updated to be $(1 - \alpha) * LRB(v) + \alpha * r$, where α is the step-size parameter whose value is empirically initialized to 0.4 and is decreased by 10^{-6} after every conflict until it reaches 0.06.

Note that LRB considers the $t_2 - t_1$ conflicts in their entirety, while VSIDS emphasizes more on the most recent conflicts because inc for VSIDS is increased after every conflict, which gives LRB stronger global characteristics and VSIDS stronger local characteristics. Because of this difference between VSIDS and LRB, they perform differently on different instances. Currently, no strategy obtains good results on all types of instances. Therefore, before solving an instance, knowing which branching strategy is better for solving it can greatly improve the performance of the solver. Unfortunately, it is hard to know which branching strategy is better for an instance before solving it. The general useful approach is to use both strategies in the solver. For example, the Maple solver uses LRB for the first half of the time limit and uses VSIDS for the second half. Another solution is to use both strategies alternatively with the phase length multiplied by a growing coefficient. Apart from the above two approaches, a reinforced learning-based Multi-Armed Bandit (MAB) framework was proposed recently to combine VSIDS and History-based Branching Heuristic (CHB)[53]. Note that CHB is the previous version of LRB without considering learning rate. It selects a proper branching strategy with Upper Confidence Bound (UCB)[55]. The solver using this technique ranked the first place in the main track of SAT Competition 2021. No matter which technique is used, the combined ones always perform better than the single branching strategy. In general, although the technique of using VSIDS and LRB for two half continuous time slots is simple and direct, it performs quite stable for most cases. And using MAB which self-adaptively chooses the branching strategy according to the conflicts production is also competitive.

4 Branching Strategy Selection and Learnt Clause Vivification Ratio

The main purpose of this paper is to explore the relationship between the learnt clause vivification ratio and the two branching heuristics of VSIDS and LRB by analyzing the characteristics of two different types of instances. The following part first introduces the characteristics of the two types of instances, and then analyzes the relationship between the vivification ratio of learnt clauses and the effectiveness of the branching strategies. Finally, we introduce the proposed approach in detail based on the above analysis, showing that the branching strategy selecting approach significantly improves the performance of the solver.

4.1 Two typical classes of SAT instances

In this section, we describe the two types of instances used in this paper: HWMCC instances generated from real-world EDA applications and PoNo instances crafted from the reduction of the multiplication of two polynomials.

4.1.1 *HWMCC instances*

In various EDA applications, one often needs to check the equivalence of two combinatorial circuits. A common way to do so is to construct a miter circuit from the two original combinatorial circuits such that the unsatisfiability of the miter circuit implies the equivalence of the two original circuits. Since each combinatorial circuit can be formulated as a Direct Acyclic Graph (DAG), in which each internal node of the DAG is one of the seven standard logical operations ($AND, OR, NOT, NAND, NOR, XOR,$ and $NXOR$), the miter circuit can be encoded into a CNF using the rules of Tseitin transformation[56] defined in Table 1.

Table 1 Encoding logical operations into CNFs

Logical operations	Equivalent CNF expression
$C = AND(A, B)$	$(\neg A \vee \neg B \vee C) \wedge (A \vee \neg C) \wedge (B \vee \neg C)$
$C = OR(A, B)$	$(A \vee B \vee \neg C) \wedge (\neg A \vee C) \wedge (\neg B \vee C)$
$C = NOT(A)$	$(\neg A \vee \neg C) \wedge (A \vee C)$
$C = NAND(A, B)$	$(\neg A \vee \neg B \vee \neg C) \wedge (A \vee C) \wedge (B \vee C)$
$C = NOR(A, B)$	$(A \vee B \vee C) \wedge (\neg A \vee \neg C) \wedge (\neg B \vee \neg C)$
$C = XOR(A, B)$	$(\neg A \vee \neg B \vee \neg C) \wedge (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (\neg A \vee B \vee C)$
$C = NXOR(A, B)$	$(\neg A \vee \neg B \vee C) \wedge (A \vee B \vee C) \wedge (A \vee \neg B \vee \neg C) \wedge (\neg A \vee B \neg C)$

The HWMCC instances used in this paper are from *SAT Competition 2017* and are provided by Armin Biere (the organizer of the Hardware Model Checking Competition[57]). Armin Biere generated 433 small-scale and 330 large-scale CNF instances from 123 real-world and 12 circuits using AIGUNROLL from the AIGER tools[58]. By analyzing these instances, 41 challenging ones have been chosen, which cannot be solved within the time limit by any existing solver before 2017. We use these 41 challenging instances to test the relationship of vivification ratio and different branching strategies between VSIDS and LRB.

4.1.2 *PoNo instances*

For comparison, we introduce the PoNo benchmark which contains 38 SAT instances encoding the problem of multiplying two polynomials of degree $n - 1$ with t ($t \leq n^2$) coefficient products. The initial objective of this encoding is to use SAT solvers to reduce the number of needed coefficient products to multiply two polynomials. We also submitted these instances to the 2017 SAT Competition[59].

A simple example of polynomial multiplication can be expressed by Equation (1):

$$(ax+b)(cx+d) = acx^2 + (ad+bc)x + bd \tag{1}$$

The trivial multiplication of two polynomials of degree 1 needs 4 coefficient products: $\{ac, ad, bc, bd\}$. A smart multiplication of the two polynomials needs only 3 coefficient products $\{ac, (a+b)(c+d), bd\}$, as expressed in Equation (2):

$$(ax+b)(cx+d) = acx^2 + ((a+b)(c+d) - ac - bd)x + bd \tag{2}$$

In Equation (2), we need more addition and subtraction operations than in Equation (1). However, multiplication is much more costly than addition and subtraction. Thus, we can multiply two polynomials of degree 1 more quickly using Equation (2) than using Equation (1).

In the sequel, we describe how to encode as a SAT instance the problem of multiplying two polynomials of degree $n-1$ using t ($t \leq n^2$) coefficient products. When the obtained SAT instance is satisfiable, the SAT solution gives a way to multiply two polynomials of degree $n-1$ using t coefficient products. When the obtained SAT instance is unsatisfiable, we know that more than t coefficient products are needed. We refer to Refs. [60, 61] for other efficient algorithms for polynomials.

Consider two polynomials of degree $n-1$:

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$$
$$B(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_1x + b_0$$

Their product is

$$A(x) \times B(x) = c_{2n-2}x^{2n-2} + c_{2n-3}x^{2n-3} + \cdots + c_1x + c_0$$

We want to compute $A(x) \times B(x)$ using t ($t \leq n^2$) coefficient products: $P_1, P_2, \cdots, P_t$, where each P_l ($1 \leq l \leq t$) is of the form $(a'_1 + a'_2 + \cdots)(b'_1 + b'_2 + \cdots)$ with $a'_1, a'_2, \cdots \in \{a_{n-1}, a_{n-2}, \cdots, a_0\}$ and $b'_1, b'_2, \cdots \in \{b_{n-1}, b_{n-2}, \cdots, b_0\}$. Addition and subtraction of these products give the coefficients c_k ($0 \leq k \leq 2n-2$) of $A(x) \times B(x)$. The problem becomes to determinate a'_i and b'_j for each product. In order to solve this problem, we first define the following Boolean variables.

- $a_{il} = 1$ iff a_i is involved in product P_l.
- $b_{jl} = 1$ iff b_j is involved in product P_l.
- $c_{kl} = 1$ iff product P_l is used to compute c_k.
- $x_{ijkl} = 1$ iff a_i and b_j are involved in product P_l, and product P_l is used to compute c_k.

We then define the clauses, which encode the following properties:

- $x_{ijkl} \equiv a_{il} \wedge b_{jl} \wedge c_{kl}$
- For each i and j ($0 \leq i, j \leq n-1$) and for each k ($0 \leq k \leq 2n-2$) such that $i + j \neq k$, if a_i and b_j are involved in product P_l (i.e., $a_{il} \wedge b_{jl}$ is implied) and P_l is used to produce c_k, then the product of a_i and b_j should be eliminated by subtraction using another product $P_{l'}$ involving a_i and b_j. If $i + j = k$, one product of a_i and b_j should remain in c_k. So,

$$\sum_{l=1}^{t} x_{ijkl} \bmod 2 = \begin{cases} 0, & \text{if } i+j=k \\ 1, & \text{otherwise} \end{cases}$$

We generated 38 SAT instances, using the encoding described above, by varying n and t. Each combination of n and t gives an instance denoted by NnTt. The constructed PoNo benchmark differs from the real-world HWMCC benchmark in many ways, which we will describe in Section 4.2.

4.2 Observation and motivation

In this section, we present our observations on the differences between HWMMC and PoNo benchmarks with respect to the learnt clause vivification ratio. We analyze the relationship between the vivification ratio and the efficiency of the branching strategies on each type of instances, which constitutes the initial motivation of our proposed approach.

4.2.1 *Redundancies and vivification ratio*

It is widely known that SAT instances often contain redundancies and eliminating this redundancy can greatly help solve the SAT instances. The HWMCC instances contain redundancy in the original CNF formula, because combinatorial circuits can contain redundant logic gates for efficiency, while the PoNo instances do not contain any redundancy in their construction. Nevertheless, when solving an instance, a CDCL solver will add learnt clauses that often contain redundant literals, even for instances that do not contain redundancy initially like for the PoNo instances.

(1) Size reduction on HWMCC

We test the Maple[62] and Maple_LCM[63] solvers on the HWMCC instances. The Maple solver is based on COMiniSatPs[64] which is created by applying a series of small diff patches to MiniSat[65]. The only difference between these two solvers is that Maple_LCM uses a learnt clause vivification named LCM simplification method, while Maple does not. Note that Maple and Maple_LCM were the winners of the main track of SAT Competition 2016 and 2017 respectively. Table 2 presents the ratio of original redundant literals (denoted as redundant_ratio) and the vivification ratio (denoted as vivi_ratio) on learnt clauses by LCM for each instance.

We use the following method to detect redundant literals in each original or learnt clause of each instance:

① set all literals in the clause to be false, except the literal to be checked.

② perform unit propagation (UP).

③ if UP produces an empty clause (conflict), the checked literal is redundant in the clause.

The ratio of redundant literals in the initial CNF formula is computed before starting the instance solving by summing up the number of all detected redundant literals in all original clauses and dividing the obtained sum by the total number of literals in these clauses, and the ratio of redundant literals in learnt clauses is similarly computed among all learnt clauses until the end of the instance solving. These redundant literals are removed from their clauses once detected, performing clause vivification. So the ratio of redundant clauses is also called vivification ratio.

We observe from Table 2 that most HWMCC instances do not have many original redundant literals (less than 1%), while the vivification ratio on the learnt clauses is significant (29.78% on average). It indicates that the performance of vivification on the learnt clauses is not related very much to the ratio of the original redundant literals. The solver Maple_LCM (with vivification on learnt clauses) outperforms Maple (without vivification) by solving 5 more instances among the 41 instances. We also observe that, for the instances containing more than 5% original redundant literals, e.g., 6s516r-k17 and 6s516r-k18, Maple_LCM uses much less time compared to Maple. The reason lies in that the learnt clauses tend to contain redundant literals if they already exist in the original clauses together with the redundant literals resulted in by the learning process, and therefore, the vivification process can simplify the clauses by eliminating these literals.

(2) Size reduction on PoNo

We solve the PoNo instances using both Maple and Maple_LCM. The results are shown in Table 3 where the CPU time, the satisfiability, and the vivification ratio for each instance are reported. The vivification does not help to solve more problem instances, but it reduces the consumed CPU time for most of the solved instances. The average time for Maple is 832.29 s but 640.47 s for Maple_LCM.

Table 2 Comparison between Maple and Maple_LCM on HWMCC

Instance	CPU time (s)		vivi_ratio (%)	original redundant_ratio (%)
	Maple	Maple_LCM		
6s105-k35.cnf	1,297.61	964.48	35.85	0.26
6s161-k17.cnf	4,141.58	2,749.23	25.45	0.43
6s161-k18.cnf	—	—	23.99	0.39
6s179-k17.cnf	3,066.29	662.4	40.43	2.38
6s188-k44.cnf	1,726.07	985.42	36.76	0.58
6s188-k46.cnf	2,285.93	1,012.72	41.58	0.55
6s33-k33.cnf	1,005.14	438.96	43.90	0.49
6s33-k34.cnf	1,019.97	710.65	35.74	0.47
6s340rb63-k16.cnf	22.58	13.98	34.51	3.73
6s340rb63-k22.cnf	326.16	199.41	22.35	2.90
6s341r-k16.cnf	223.36	83.82	32.86	1.43
6s341r-k19.cnf	1,218.09	297.46	28.69	1.17
6s366r-k72.cnf	2,031.58	929.27	39.92	0.07
6s399b02-k02.cnf	—	—	15.06	0.02
6s399b03-k02.cnf	—	—	14.00	0.02
6s44-k38.cnf	1,594.35	772.21	33.54	1.54
6s44-k40.cnf	2,026.48	1,286.06	34.89	1.51
6s516r-k17.cnf	2,439.69	1,368.93	41.24	5.65
6s516r-k18.cnf	4,967.21	2,715	42.83	5.43
beembkry8b1-k45.cnf	1,670.45	725.82	35.35	0.14
beemcmbrdg7f2-k32.cnf	—	1,961.92	33.93	0.32
beemfwt4b1-k48.cnf	—	—	37.91	0.43
beemhanoi4b1-k32.cnf	—	3,244.89	35.19	0.16
beemhanoi4b1-k37.cnf	—	—	29.71	0.14
beemlifts3b1-k29.cnf	1,419.55	957.29	45.08	2.80
beemloyd3b1-k31.cnf	1,992.58	1,225.17	35.77	1.51
bob12s02-k16.cnf	—	—	4.07	0.00
bob12s02-k17.cnf	—	—	4.91	0.00
bobpcihm-k30.cnf	—	4,907.19	19.04	2.39
bobpcihm-k31.cnf	—	—	19.12	2.28
bobpcihm-k32.cnf	—	—	19.54	2.17
bobpcihm-k33.cnf	—	—	20.27	2.07
intel032-k84.cnf	4,240.67	855.38	31.71	0.66
intel065-k11.cnf	—	—	17.37	2.76
intel066-k10.cnf	—	—	16.04	3.02
oski15a10b06s-k24.cnf	—	—	15.58	2.44
oski15a10b08s-k23.cnf	—	4,395.33	22.70	2.79
oski15a10b10s-k20.cnf	—	3,055.18	28.02	2.75
oski15a10b10s-k22.cnf	—	—	21.12	2.58
oski15a14b04s-k16.cnf	3,328.91	368.31	66.03	1.20
oski15a14b30s-k24.cnf	3,189.68	1,929.16	40.20	0.80
AVG	2,056.09	1,437.62	29.81	1.52

We can observe that the vivification ratio for the UNSAT instances is higher than for the SAT instances. Among the UNSAT instances, the ratio is higher for those with a smaller t value. The reason may be that in the UNSAT instances, there are more tight constraints, expecially when t is small.

We also find another characteristic that for the instances with the same n value, they become SAT from UNSAT as the t value increases. At the same time, the instance size grows significantly. Here, we test the vivification ratio on the instances with $n = 5$ and $t = \{5, 6, \cdots, 18\}$ of which the results are shown in Table 4. The results are consistent with the analysis that the vivification ratio decreases from 21% to 3% as t increases from 6 to 18.

Table 3 Comparison between Maple and Maple_LCM on PoNo

Instance	Maple		Maple_LCM		
	CPU time (s)	Satisfiability	CPU time (s)	Satisfiability	vivi_ratio (%)
N5T06.cnf	23.98	UNSAT	60.69	UNSAT	21.37
N5T07.cnf	—	—	—	—	9.29
N5T14.cnf	—	—	1,065.81	SAT	4.62
N5T15.cnf	930.14	SAT	—	—	4.25
N5T16.cnf	39.07	SAT	92.15	SAT	3.84
N6T06.cnf	37.29	UNSAT	35.86	UNSAT	23.10
N6T07.cnf	4,646.38	UNSAT	3,395.28	UNSAT	14.72
N6T25.cnf	995.03	SAT	672.38	SAT	2.63
N6T27.cnf	198.78	SAT	99.24	SAT	2.74
N6T28.cnf	250.19	SAT	174.44	SAT	2.79
N6T29.cnf	1,622.49	SAT	165.76	SAT	2.63
N6T30.cnf	100.87	SAT	99.94	SAT	2.69
N7T07.cnf	—	—	—	—	10.34
N7T08.cnf	—	—	—	—	7.26
N7T42.cnf	—	—	—	—	1.98
N7T43.cnf	—	—	—	—	1.92
N7T44.cnf	—	—	—	—	1.88
N7T45.cnf	—	—	—	—	1.92
N7T46.cnf	—	—	—	—	1.81
N8T60.cnf	—	—	—	—	1.53
N8T61.cnf	—	—	—	—	1.50
N8T62.cnf	—	—	—	—	1.46
N8T63.cnf	—	—	—	—	1.51
N11T118.cnf	—	—	—	—	1.11
N13T165.cnf	—	—	—	—	0.85
N13T166.cnf	—	—	—	—	0.92
N14T194.cnf	—	—	—	—	0.84
N27T6.cnf	153.77	UNSAT	230.33	UNSAT	21.65
N29T6.cnf	176.28	UNSAT	207.07	UNSAT	24.73
N37T6.cnf	514.25	UNSAT	415.75	UNSAT	22.73
N39T6.cnf	618.62	UNSAT	581.28	UNSAT	22.86
N42T6.cnf	695.86	UNSAT	555.21	UNSAT	26.05
N44T6.cnf	723.79	UNSAT	890.66	UNSAT	21.43
N45T6.cnf	654.03	UNSAT	727.18	UNSAT	20.98
N49T6.cnf	1,095.61	UNSAT	819.04	UNSAT	22.39
N51T6.cnf	1,379.78	UNSAT	1,082.04	UNSAT	21.09
N52T6.cnf	1,332.57	UNSAT	1,036.91	UNSAT	26.48
N54T6.cnf	1,289.34	UNSAT	1,042.88	UNSAT	22.41
AVG	832.29	—	640.47	—	10.11%

(3) Conflict index, UP index and vivification ratio

We observe that, during the search process, the implication graphs of the HWMCC instances are much more complex than those of the PoNo ones. So, we define two indicators to reflect the differences in the characteristics of these two benchmarks.

Definition 1 (Conflict Index). It is the average number of conflict level literals involved in a conflict, that is, the number of active literals.

Definition 2 (UP Index). It is the average number of literals propagated by UP after

Table 4 The vivification ratio on instances with $n = 5, t = \{5, 6, \cdots, 18\}$

Instance	Maple_LCM		
	CPU time (s)	Satisfiability	vivi_ratio (%)
N5T05.cnf	2.34	UNSAT	16.67
N5T06.cnf	61.31	UNSAT	21.37
N5T07.cnf	—	—	9.26
N5T08.cnf	—	—	6.47
N5T09.cnf	—	—	6.23
N5T10.cnf	—	—	5.53
N5T11.cnf	—	—	5.27
N5T12.cnf	—	—	4.87
N5T13.cnf	—	—	4.51
N5T14.cnf	1,064.59	SAT	4.62
N5T15.cnf	—	—	4.24
N5T16.cnf	90.38	SAT	3.84
N5T17.cnf	81.48	SAT	3.44
N5T18.cnf	24.89	SAT	3.50

branching on a decision literal.

Table 5 and Table 6 show the relationship between the conflict index, UP index and vivification ratio of the HWMCC and PoNo instances, respectively.

Compared with the UP index, the conflict index and the vivification ratio are even more consistent. For example, for the instances bob12s02-k16.cnf and bob12s02-k17.cnf, the conflict indexes respectively are 2.33 and 2.27, which are much lower than those of other HWMCC instances. The vivification ratios of these two instances are 4.07% and 4.91%, which are also far lower than those of other HWMCC instances. In addition, for the instances oski15a14b04s-k16.cnf and oski15a14b30s-k24.cnf, the conflict indexes are 10.46 and 11.57 respectively. The vivification ratios of these two instances are 66.03% and 40.2%, which are also greater than those of other HWMCC instances.

4.2.2 *Branching strategies and vivification ratio*

The LBD of a learnt clause is roughly the number of decisions needed to produce a conflict by UP. Table 7 shows the LBD and size of the learnt clauses when solving the HWMCC instances and the PoNo instances using VSIDS and LRB strategies, respectively. It also presents the number of solved instances by Maple_LCM only using one branching strategy between VSIDS (denoted by Maple_LCM/VSIDS) and LRB (denoted by Maple_LCM/LRB). Note that Maple_LCM uses LRB for the first half time and VSIDS for the second half time.

We can make the following two observations from Table 7.

(1) The LBD of the learnt clauses (denoted as learnts_LBD) in the HWMCC instances is much smaller than that in PoNo, i.e., the number of decisions needed to produce a conflict is much smaller when solving the HWMCC instances than when solving the PoNo instances, because the UP index and the conflict index of the HWMCC instances are much greater than those of the PoNo instances. In fact, the average LBD of HWMCC instances from the Maple_LCM/VSIDS solver is 9.59, while this value for PoNo is 25.06. When vivifing a learnt clause $l_1 \vee l_2 \vee \cdots \vee l_k$ by successively propagating $\neg l_1, \neg l_2, \cdots, \neg l_i (i < k)$, these negated literals can be considered as decisions, and fewer negative literals are needed to be propagated to produce a conflict in the HWMCC instances than in the PoNo instances. For the vivified learnt clauses which are selected by the LCM approach from the learnt clause database with smaller LBD value, we can see that the size of vivified learnt clauses (denoted as vivi_Size) in the HWMCC instances is greater than the one in the PoNo instances; on the other hand, vivified learnt clauses in HWMCC and PoNo have roughly the same LBD value (denoted as vivi_LBD). Note that all the above values are average. So, simplifying the learnt clauses is usually much easier for the HWMCC instances

Table 5 Conflict index, UP index and vivification ratio on HWMCC benchmark

Instance	Conflict index	UP index	vivi_ratio (%)
6s105-k35.cnf	6.88	52.29	35.85
6s161-k17.cnf	6.65	41.61	25.45
6s161-k18.cnf	7.08	42.29	23.99
6s179-k17.cnf	12.45	20.08	40.43
6s188-k44.cnf	7.82	182.82	36.76
6s188-k46.cnf	7.57	190.39	41.58
6s33-k33.cnf	10.19	12.44	43.90
6s33-k34.cnf	10.74	12.46	35.74
6s340rb63-k16.cnf	5.03	48.21	34.51
6s340rb63-k22.cnf	5.57	88.04	22.35
6s341r-k16.cnf	7.46	236.65	32.86
6s341r-k19.cnf	7.03	260.78	28.69
6s366r-k72.cnf	8.66	619.66	39.92
6s399b02-k02.cnf	9.66	9.32	15.06
6s399b03-k02.cnf	8.35	9.43	14.00
6s44-k38.cnf	7.38	87.38	33.54
6s44-k40.cnf	6.61	87.3	34.89
6s516r-k17.cnf	9	41.91	41.24
6s516r-k18.cnf	6.57	37.03	42.83
beembkry8b1-k45.cnf	22.92	10.93	35.35
beemcmbrdg7f2-k32.cnf	17.03	28.62	33.93
beemfwt4b1-k48.cnf	40.75	33.8	37.91
beemhanoi4b1-k32.cnf	13.97	53.11	35.19
beemhanoi4b1-k37.cnf	11.26	52.78	29.71
beemlifts3b1-k29.cnf	35.16	43.5	45.08
beemloyd3b1-k31.cnf	9.68	40.24	35.77
bob12s02-k16.cnf	2.33	99.6	4.07
bob12s02-k17.cnf	2.27	111.62	4.91
bobpcihm-k30.cnf	8.77	78.29	19.04
bobpcihm-k31.cnf	7.81	84.25	19.12
bobpcihm-k32.cnf	8.52	78.83	19.54
bobpcihm-k33.cnf	8.19	78.77	20.27
intel032-k84.cnf	7.27	30.04	31.71
intel065-k11.cnf	2.97	19.21	17.37
intel066-k10.cnf	3.28	39.54	16.04
oski15a10b06s-k24.cnf	5.11	22.83	15.58
oski15a10b08s-k23.cnf	4.83	22.38	22.70
oski15a10b10s-k20.cnf	4.94	22.42	28.02
oski15a10b10s-k22.cnf	4.85	22.43	21.12
oski15a14b04s-k16.cnf	10.46	152.23	66.03
oski15a14b30s-k24.cnf	11.57	109.88	40.20
AVG	9.62	80.86	29.81

than for the PoNo instances, explaining the higher vivification ratio for the HWMCC instances.

(2) The VSIDS branching strategy is much better than the LRB branching strategy for the HWMCC instances, while the LRB branching strategy is much better than the VSIDS one for the PoNo instances.

The above two observations suggest that VSIDS should be more used for instances on which the learnt clause vivification ratio is large, and LRB should be more used for instances on which the learnt clause vivification ratio is small. However, some instances with high vivification ratio may also be crafted instances. For example, the 3 instances which are crafted_n10_d6_c3_num18.cnf, crafted_n11_d6_c4_num19.cnf and crafted_n12_d6_c4_num9.cnf from the main track of the 2020 SAT competition have very high vivification ratios. We test these instances with the two solvers Maple_LCM/VSDIS and

Maple_LCM/LRB. It can been seen that the performance of only using the LRB branching strategy is much better than only using the VSIDS one (as shown in Table 8).

Table 6 Conflict index, UP index and vivification ratio on partial PoNo benchmark

Instance	Conflict index	UP index	vivi_ratio (%)
N5T06.cnf	4.85	5.14	21.37
N5T07.cnf	4.73	4.81	9.29
N5T14.cnf	2.75	3.76	4.62
N5T15.cnf	3.11	3.64	4.25
N5T16.cnf	3.15	3.8	3.84
N6T06.cnf	4.88	7.44	23.10
N6T07.cnf	4.33	6.79	14.72
N6T25.cnf	2.42	1.49	2.63
N6T27.cnf	2.41	1.49	2.74
N6T28.cnf	2.63	1.5	2.79
N6T29.cnf	2.54	1.49	2.63
N6T30.cnf	2.5	1.49	2.69
N7T07.cnf	4.27	9.73	10.34
N7T08.cnf	3.82	9.65	7.26
N7T42.cnf	2.22	1.5	1.98
N7T43.cnf	2.35	1.5	1.92
N7T44.cnf	2.44	1.5	1.88
N7T45.cnf	2.47	1.5	1.92
N7T46.cnf	2.21	1.49	1.81
AVG	3.16	3.66	6.40

Table 7 LBD of the learnt clauses of two benchmarks using VSIDS and LRB strategies

Instance	HWMCC		PoNo	
	Maple_LCM/VSIDS	Maple_LCM/LRB	Maple_LCM/VSIDS	Maple_LCM/LRB
learnts_LBD	9.59	14.39	25.06	56.81
learnts_size	31.74	52.14	37.45	80.49
vivi_LBD	3.93	3.98	4.31	4.42
vivi_size	16.85	15.82	9.37	8.53
vivi_ratio (%)	34.91	28.92	7.46	7.44
nbSloved	**32**	19	13	**21**

Table 8 Results of crafted instances using VSIDS and LRB strategies

Instance	Maple_LCM/VSIDS		Maple_LCM/LRB	
	CPU time (s)	vivi_ratio (%)	CPU time (s)	vivi_ratio (%)
crafted_n10_d6_c3_num18.cnf	—	34.17	331.26	31.25
crafted_n11_d6_c4_num19.cnf	976.18	38.60	203.43	36.20
crafted_n12_d6_c4_num9.cnf	542.17	45.92	226.82	41.34

These instances which are typical crafted ones are encoded through representing the graph isomorphism of two graphs. The original problems corresponding to these instances are obvious UNSAT problems. When they are encoded into SAT instances, the redundancy and the vivification ratio are significantly high in searching process. This phenomenon can also be seen in the PoNo instance N27T6.cnf. Although the vivification ratio of N27T6.cnf is more than 20%, it is still a crafted one. Therefore, in this paper we consider to use more LRB branching strategy for the instances with low vivification ratio. We will propose an approach in this direction in the next section.

4.3 Branching strategy selection based on the vivification ratio

The searching procedure of the CDCL SAT solver is shown in Algorithm 1. The SAT solver performs the preprocessing of the instance firstly, including deleting variables, clauses and literals. Then during search, the learnt clause vivification will be performed at certain beginning of restarts. In the main searching procedure, unit propagation is performed firstly. If conflicts occur, conflict analysis is triggered, a new learnt clause is generated, based on which the solver backtracks. If there is no conflict, the solver selects a suitable variable to assign a truth value using a branching strategy. This process is repeated until a conflict is produced in the 0th level to prove unsatisfiability, or an assignment satisfying all clauses.

In this section, we describe our approach to allow Algorithm 1 to choose a suitable branching strategy based on vivification ratio during search, by calling the function chooseBranch*(viviRatio)* (described in Algorithm 2).

Algorithm 1. CDCL (ϕ), a generic CDCL SAT algorithm

```
Input: $\phi$: A CNF formula with original and learnt clauses
Output: SATISFIABLE or UNSATISFIABLE
1.  begin
2.      $\phi \leftarrow$ preprocessing ($\phi$):
3.      while true do
4.          $S \leftarrow$ chooseBranch (viviRatio); /* call Algorithm 2 to choose a branching strategy*/
5.          currentLevel $\leftarrow$ 0; /* start or restart search */
6.          $\phi \leftarrow$ vivification ($\phi$); /* select some learnt clauses to vivify */
7.          while true do
8.              $cl \leftarrow$ UP($\phi$); /* all variables assigned by UP are recorded with currentLevel */
9.              if $cl$ is a falsified clause then
10.                 if currentLevel == 0 then
11.                     return UNSATISFIABLE;
12.                 else
13.                     newLearntClause $\leftarrow$ analyze ($cl$); /* conflict analysis to learn a new clause
                          */
14.                     level $\leftarrow$ the second highest level in newLearntClause;
15.                     backtrackTo (level); /* cancel all variable assignments higher than level level
                          */
16.                     currentLevel $\leftarrow$ level;
17.                 end if
18.             else
19.                 if all variables are assigned then
20.                     return SATISFIABLE;
21.                 else if restart condition is satisfied then
22.                     backtrackTo(0); /* cancel all assignments depending on a decision */
23.                     break; /* restart */
24.                 else if learnt clause database reduction condition is satisfied then
25.                     remove a subset of learnt clauses;
26.                 else
27.                     currentLevel $++$;
28.                     $x \leftarrow$ a non-assigned variable selected according to the strategy $S$;
29.                     add the unit clause $x$ or $\neg x$ into $\phi$ according to a polarity heuristic such as
                          phase saving;
30.                 end if
31.             end if
32.         end while
33.     end while
34. end
```

Concretely, the purpose of the chooseBranch*(viviRatio)* function is to determine a probability P_b to select the LRB or VSIDS branching strategy, depending on the vivification ratio of learnt clauses in the last γ restarts since the last execution of the chooseBranch*(viviRatio)* function or from the beginning, where γ is a parameter initialized to 10,000 and is increased by 10% each time P_b is re-evaluated. When the learnt clause vivification ratio is lower than α, where α is a parameter fixed to 8%, then P_b is set to 0.8 to select the LRB branching strategy. Otherwise, P_b is set to 0.5. Finally Algorithm 2 returns the LRB strategy with probability P_b and the VSIDS strategy with probability $1 - P_b$.

Algorithm 2. chooseBranch *(viviRatio)*, choose a branching strategy

```
Input: viviRatio: vivification ratio of learnt clauses
Output: LRB or VSIDS branching strategy
1.  begin
2.      if the number of restarts since the last execution of this function exceeds $\gamma$ then
3.          if viviRatio $< \alpha$ then
4.              $P_b \leftarrow \beta$; /* $\alpha = 8\%$ and $\beta = 0.8$ */
5.          else
6.              $P_b \leftarrow 0.5$;
7.          end if
8.          $\gamma \leftarrow \gamma + \gamma/10$;
9.          with probability $P_b$, return the LRB strategy; else return the VSIDS strategy;
10.     end if
11. end
```

5 Experiments

In this section, we evaluate our proposed approach on several SAT Competition benchmarks.

5.1 Experimental protocol

The test suite includes the instances from the main tracks (application + crafted) of the SAT Competition 2017, 2018 and 2020. The experiments were performed on Intel Xeon E5-2680 v4 processors with 2.40 GHz and 20 GB of memory under Linux. The cutoff time is 5,000 s for each solver and each instance, including the preprocessing time and the search time, unless otherwise stated. For each solver and each benchmark, we report the number of solved SAT/UNSAT instances and total solved instances, denoted as “#SAT”, “#UNSAT” and “#Solved”respectively, and the penalized run time “PAR2” (as used in SAT Competitions), where the run time of a failed run is penalized as twice the cutoff time.

The tested approach is implemented based on the SAT solver Maple_CM which is an improved version of the Maple_LCM solver. The difference with Maple_LCM is that the Maple_CM solver uses vivification for more in-depth clause simplification. In fact, Maple_CM not only simplifies learnt clauses more than once using vivification, but also simplifies original clauses during pre-processing and in-processing. Note that the performance of Maple_CM is greatly improved compared to Maple_LCM, and it won the third place in the main track of the 2018 SAT competition.

5.2 Efficiency analysis

We implemented Algorithm 2 on top of the solver Maple_CM. The resulting solver is named Maple_CM+. Besides, we created the solver Maple_CM+/$P_b0.5$ by setting the parameter P_b to the fixed value 0.5 (set β as 0.5 in Algorithm 2). In other words, Maple_CM+/$P_b0.5$ is Maple_CM+ except that it selects LRB and VSIDS with the same probability.

Table 9 compares the solvers Maple_CM, Maple_CM+/$P_b0.5$ and Maple_CM+ on the instances from the main tracks of the SAT Competition (noted as SC) 2017, 2018 and 2020. The proposed Branching Strategy Selection (BSS) approach obviously improves the performance of all solvers on instances of both SC2018 and SC2020. In particular, for instances of SC2020, Maple_CM+ can solve 18 instances more than Maple_CM and the PAR2 solution time is also greatly reduced with the help of the BSS approach. Moreover, we can see that Maple_CM+ performs consistently better than Maple_CM+/$P_b0.5$ for all instances of every SAT competition.

Table 9 Comparison of the branching strategy selection on different reduction ratios

Instances	Solver	#Total	#SAT	#UNSAT	PAR2 (s)
SC2020 (400)	Maple_CM	196	90	106	5,672.11
	Maple_CM+/$P_b0.5$	219	108	111	5,022.82
	Maple_CM+	**224**	**111**	**113**	**5,006.07**
SC2018 (400)	Maple_CM	229	129	100	4,643.99
	Maple_CM+/$P_b0.5$	233	133	100	4,653.62
	Maple_CM+	**237**	**136**	**101**	**4,536.32**
SC2017 (350)	Maple_CM	**227**	**110**	**117**	**4,116.99**
	Maple_CM+/$P_b0.5$	218	101	117	4,313.60
	Maple_CM+	224	108	116	4,163.75

However, the performance of Maple_CM+ is worse than that of Maple_CM for the SC2017 instances. The main reason lays on the fact that Maple_CM uses LRB for the first 2,500 seconds and VSIDS for the last 2,500 seconds. Instead, Maple_CM+ selects LRB or VSIDS with a certain probability after a series of conflicts. Moreover there are many application instances in SC2017, which are more suitable for the long-term use of the VSIDS branching strategy to solve them. Therefore, the solving results of Maple_CM are better compared to Maple_CM+ in this special case. On the other hand, for the SC2017 instance, Maple_CM+ can solve 6 more instances than Maple_CM+/$P_b0.5$, which shows that the method proposed in this paper is still effective in terms of the choice of branching strategies.

5.3 Robustness analysis

Here we tested Maple_CM+ by fixing P_b from 0% to 100% at a growth rate of 10% (i.e. set β from 0% to 100% in Algorithm 2). Note that $P_b = 0\%$ means that Maple_CM+ only uses the VSIDS branching strategy, while $P_b = 100\%$ means that Maple_CM+ only uses the LRB branching strategy. For every value of P_b, we ran Maple_CM+ to solve the instances with low vivification ratio ($vivi\_ratio < 8\%$) in SC2020, SC2018 and SC2017. Note that there are 134, 189 and 164 instances with low vivification ratio in SC2020, SC2018 and SC2017, respectively.

Table 10 shows that the performance of the solver only using LRB branching strategy is substantially better than the performance of the solver only using VSIDS. And with the fixed value of P_b increasing gradually, the result of Maple_CM+/P_b is also continuously improved. When P_b is set to 0.7 or 0.8, the result roughly reaches the highest point. The results show that LRB should be more used for instances on which the learnt clause vivification ratio is small.

We can also see from Table 9 and Table 10 that, in general, BSS approach which is implemented in CDCL SAT solvers can significantly improve the results of SAT instances, while the results of UNSAT instances are not improved obviously. The main reason for this phenomenon is that the BSS approach is applied for crafted instances, and most of the crafted instances are SAT ones.

6 Conclusions and Future Work

We defined a new branching strategy selection approach based on the vivification ratio of learnt clauses. Through experimental investigation, we analyzed the relationship between different types of instances with the vivification ratio and branching strategy, and found that if the vivification ratio of the instance is very low, then it is more suitable for the instance to be solved by using more the LRB branching strategy. Furthermore, we performed an in-depth empirical analysis that showed that the proposed branching strategy selection approach is robust

Table 10 Robustness analysis

Instances	Solver	#Total	#SAT	#UNSAT	PAR2 (s)
SC2020 (vivi_low 134)	Maple_CM+/P_b0	41	16	25	7,450.86
	Maple_CM+/P_b0.1	49	24	25	6,932.79
	Maple_CM+/P_b0.2	51	26	25	6,784.91
	Maple_CM+/P_b0.3	50	23	27	6,806.67
	Maple_CM+/P_b0.4	49	24	25	6,860.51
	Maple_CM+/P_b0.5	54	27	27	6,463.03
	Maple_CM+/P_b0.6	55	29	26	6,397.19
	Maple_CM+/P_b0.7	58	31	27	6,296.81
	Maple_CM+/P_b0.8	**58**	**32**	26	**6,251.66**
	Maple_CM+/P_b0.9	58	31	27	6,284.46
	Maple_CM+/P_b1	57	29	**28**	6,266.66
SC2018 (vivi_low 189)	Maple_CM+/P_b0	46	32	14	7,801.30
	Maple_CM+/P_b0.1	59	45	14	7,168.91
	Maple_CM+/P_b0.2	65	50	15	6,883.54
	Maple_CM+/P_b0.3	74	58	16	6,511.65
	Maple_CM+/P_b0.4	70	55	15	6,541.22
	Maple_CM+/P_b0.5	70	55	15	6,540.55
	Maple_CM+/P_b0.6	72	56	16	6,494.40
	Maple_CM+/P_b0.7	78	62	16	6,213.59
	Maple_CM+/P_b0.8	75	58	17	6,382.57
	Maple_CM+/P_b0.9	73	56	**17**	6,393.49
	Maple_CM+/P_b1	**81**	**65**	16	**6,102.91**
SC2017 (vivi_low 164)	Maple_CM+/P_b0	46	28	18	7,590.08
	Maple_CM+/P_b0.1	57	38	19	6,976.40
	Maple_CM+/P_b0.2	57	38	19	6,894.34
	Maple_CM+/P_b0.3	53	34	19	7,072.37
	Maple_CM+/P_b0.4	56	37	19	6,951.51
	Maple_CM+/P_b0.5	62	42	20	6,678.19
	Maple_CM+/P_b0.6	61	41	20	6,686.56
	Maple_CM+/P_b0.7	**73**	**52**	21	**6,189.76**
	Maple_CM+/P_b0.8	68	46	22	6,366.73
	Maple_CM+/P_b0.9	69	46	**23**	6,316.76
	Maple_CM+/P_b1	62	40	22	6,652.67

and allows to solve more instances from recent SAT competitions, especially for the main track of the 2020 SAT competition.

In fact, through the analysis of the redundancy of the original clauses for the crafted and application instances, it can be found that the original clauses of the application instances have a higher redundancy ratio. In other words, using the vivification method can detect many redundant literals of original clauses during pre-processing for application instances. Instead, generally speaking, the original clauses redundancy ratio of crafted instances is 0.

Therefore, in future work, the redundancy ratio of original clauses can also be considered as a measure to analyze the types of instances. Specifically, if the redundancy ratio of the original clauses of an instance is 0, the instance is a crafted one, even if the vivification ratio of the learnt clauses of the instance is very high. So for an instance, if the learnt vivification ratio is very high and the original redundancy ratio is also high, we can consider it as an application instance and use more VSIDS for solving it. In summary, the original clauses redundancy ratio and the learnt clause vivification ratio can be used as two parameters to judge the type of the given instance more accurately, thereby guiding the selection of branching strategies.

References

[1] Brand D. Redundancy and don't cares in logic synthesis. IEEE Transactions on Computers, 1983, 32(10): 947–952. [doi: 10.1109/TC.1983.1676139]

[2] Cook B, Kroening D, Sharygina N. Symbolic model checking for asynchronous boolean programs. International SPIN Workshop on Model Checking of Software. Springer. 2005. 75–90. [doi: 10.1007/11537328_9]

[3] Prasad MR, Biere A, Gupta A. A survey of recent advances in SAT-based formal verification. International Journal on Software Tools for Technology Transfer, 2005, 7(2): 156–173. [doi: 10.1007/s10009-004-0183-4]

[4] Drechsler R, Fey G. Automatic test pattern generation. International School on Formal Methods for the Design of Computer, Communication and Software Systems. 2006. 30–55.

[5] Khomenko V, Koutny M, Yakovlev A. Logic synthesis for asynchronous circuits based on STG unfoldings and incremental SAT. Fundamenta Informaticae, 2006, 70(1,2): 49–73. [doi: 10.5555/2367636.2367644]

[6] Jackson D, Schechter I, Shlyahter H. Alcoa: The alloy constraint analyzer. Proc. of the 22nd international conference on Software engineering. 2000. 730–733. [doi: 10.1109/ICSE.2000.870482]

[7] Clarke E, Kroening D, Lerda F. A tool for checking ANSI-C programs. International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2004. 168–176. [doi: 10.1007/978-3-540-24730-2_15]

[8] Khurshid S, Marinov D. TestEra: Specification-based testing of Java programs using SAT. Automated Software Engineering, 2004, 11(4): 403–434. [doi: 10.1023/B:AUSE.0000038938.10589.b9]

[9] Kautz H, Selman B, *et al.* Planning as satisfiability. Proc. of the 10th European Conference on Artificial Intelligence (ECAI 92). Vienna, Austria. 1992, 92. 359–363.

[10] Rintanen J, Heljanko K, Niemelä I. Planning as satisfiability: parallel plans and algorithms for plan search. Artificial Intelligence, 2006, 170(12–13): 1031–1080. [doi: 10.1016/j.artint.2006.08.002]

[11] Lynce I, Marques-Silva J. Efficient haplotype inference with Boolean satisfiability. Proc. of National Conference on Artificial Intelligence (AAAI). 2006. 104–109.

[12] Ganzinger H, Hagen G, Nieuwenhuis R, *et al.* DPLL(T): Fast decision procedures. International Conference on Computer Aided Verification. 2004. 175–188. [doi: 10.1007/978-3-540-27813-9_14]

[13] Cimatti A. Beyond boolean sat: Satisfiability modulo theories. 2008 9th International Workshop on Discrete Event Systems. 2008. 68–73. [doi: 10.1109/WODES.2008.4605924]

[14] Li CM, Manya F, Planes J. New inference rules for Max-SAT. Journal of Artificial Intelligence Research, 2007, 30: 321–359.

[15] Li CM, Manya F, Quan Z, *et al.* Exact minsat solving. International Conference on Theory and Applications of Satisfiability Testing. 2010. 363–368.

[16] Heule MJH, Kullmann O, Marek VW. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. International Conference on Theory and Applications of Satisfiability Testing. 2016. 228–245. arXiv:1605.00723.

[17] Heule M, van Maaren H. Look-ahead based SAT solvers. Handbook of satisfiability, 2009, 185: 155–184. [doi: 10.3233/978-1-58603-929-5-155]

[18] Heule M, Dufour M, van Zwieten J, *et al.* March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. International Conference on Theory and Applications of Satisfiability Testing. 2004. 345–359. [doi: 10.1007/11527695_26]

[19] Heule MJH, Kullmann O, Wieringa S, *et al.* Cube and conquer: Guiding CDCL SAT solvers by lookaheads. Haifa Verification Conference. 2011. 50–65.

[20] Cai SW, Luo C, Su KL. Scoring functions based on second level score for k-SAT with long clauses. Journal of Artificial Intelligence Research, 2014, 51: 413–441.

[21] Luo C, Cai SW, Wu W, *et al.* Double configuration checking in stochastic local search for satisfiability. Proc. of 28h AAAI Conference on Artificial Intelligence. 2014. 2703–2709.

[22] Cai SW, Luo C, Su KL. CCAnr: A configuration checking based local search solver for non-random satisfiability. International Conference on Theory and Applications of Satisfiability Testing. 2015. 1–8.

[23] Luo C, Cai SW, Wu W, *et al.* CCLS: An efficient local search algorithm for weighted maximum satisfiability. IEEE Transactions on Computers, IEEE, 2014, 64(7): 1830–1843.

[24] Luo C, Hoos H, Cai SW. PbO-CCSAT: Boosting local search for satisfiability using programming by optimisation. International Conference on Parallel Problem Solving from Nature. 2020. 373–389.

[25] Cai SW, Zhang XD. Deep cooperation of CDCL and local search for SAT. International Conference on Theory and Applications of Satisfiability Testing. 2021.64–81.

[26] Fréchette A, Newman N, Leyton-Brown K. Solving the station repacking problem. Proc. 30th AAAI Conference on Artificial Intelligence. 2016.

[27] Mézard M, Parisi G, Zecchina R. Analytic and algorithmic solution of random satisfiability problems. Science, 2002, 297(5582): 812–815. [doi: 10.1126/science.1073287]

[28] Braunstein A, Mézard M, Zecchina R. Survey propagation: An algorithm for satisfiability. Random Structures & Algorithms, 2005, 27(2): 201–226. arXiv:cs/0212002.

[29] Kroc L, Sabharwal A, Selman B. Survey propagation revisited. arXiv:1206.5273, 2012.

[30] Gableske O. On the interpolation between product-based message passing heuristics for SAT. International Conference on Theory and Applications of Satisfiability Testing. 2013. 293–308. [doi: 10.1007/978-3-642-39071-5_22]

[31] Gableske O. An ising model inspired extension of the product-based MP framework for SAT. International Conference on Theory and Applications of Satisfiability Testing. 2014. 367–383. [doi: 10.1007/978-3-319-09284-3_27]

[32] Gableske O. Solver description of dimetheus v. 1.700 for the SAT competition 2013. Proceedings of SAT Competition 2013. 2013. 30.

[33] Marques-Silva JP, Sakallah KA. GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers, 1999, 48(5):506–521. [doi: 10.1109/12.769433]

[34] Moskewicz MW, Madigan CF, Zhao Y, *et al.* Chaff: Engineering an efficient SAT solver. Proc. of the 38th Annual Design Automation Conference. 2001. 530–535. [doi: 10.1145/378239.379017]

[35] Eén N, Biere A. Effective preprocessing in SAT through variable and clause elimination. International Conference on Theory and Applications of Satisfiability Testing. Springer. 2005. 61–75. [doi: 10.1007/11499107_5]

[36] Bacchus F, Winter J. Effective preprocessing with hyper-resolution and equality reduction. International Conference on Theory and Applications of Satisfiability Testing. Springer. 2003. 341–355.

[37] Järvisalo M, Heule MJH, Biere A. Inprocessing rules. International Joint Conference on Automated Reasoning. Springer. 2012. 355–370. [doi: 10.1007/978-3-642-31365-3_28]

[38] Beame P, Kautz H, Sabharwal A. Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research, 2004, 22:319–351. [doi: 10.1613/jair.1410]

[39] Sörensson N, Biere A. Minimizing learned clauses. International Conference on Theory and Applications of Satisfiability Testing. Springer. 2009. 237–243. [doi:10.1007/978-3-642-02777-2_23]

[40] Han HJ, Somenzi F. On-the-fly clause improvement. International Conference on Theory and Applications of Satisfiability Testing. Springer. 2009. 209–222. [doi:10.1007/978-3-642-02777-2_21]

[41] Hamadi Y, Jabbour S, Sais L. Learning for dynamic subsumption. International Journal on Artificial Intelligence Tools, 2010, 19(4): 511–529. [doi: 10.1142/S0218213010000303]

[42] Luo M, Li CM, Xiao F, Manya F, Lü ZP. An effective learnt clause minimization approach for CDCL SAT solvers. Proc. of the 26th International Joint Conference on Artificial Intelligence. 2017. 703–711.

[43] Li CM, Xiao F, Luo M, *et al.* Clause vivification by unit propagation in CDCL SAT solvers. Artificial Intelligence, 2020, 279: 103197. [doi: 10.1016/j.artint.2019.103197]

[44] Freeman JW. Improvements to propositional satisfiability search algorithms [Ph.D. thesis]. University of Pennsylvania. 1995.

[45] Pretolani D. Efficiency and stability of hypergraph SAT algorithms. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1996, 26: 479–498.

[46] Jeroslow RG, Wang JC. Solving propositional satisfiability problems. Annals of Mathematics and Artificial Intelligence, 1990, 1(1–4): 167–187.

[47] Marques-Silva J. The impact of branching heuristics in propositional satisfiability algorithms. Portuguese Conference on Artificial Intelligence. Springer. 1999. 62–74. [doi:10.1007/3-540-48159-1_5]

[48] Li CM, Anbulagan A. Heuristics based on unit propagation for satisfiability problems. Proc. of the 15th International Joint Conference on Artifical Intelligence. 1997. 366–371.

[49] Liang JH, Ganesh V, Poupart P, *et al.* Learning rate based branching heuristic for SAT solvers. International Conference on Theory and Applications of Satisfiability Testing. Springer. 2016. 123–140. [doi: 10.1007/978-3-319-40970-2_9]

[50] Audemard G, Simon L. Predicting learnt clauses quality in modern SAT solvers. Proc. 21st International Joint Conference on Artificial Intelligence. 2009. 399–404.

[51] Ryan L. Efficient algorithms for clause-learning SAT solvers [Master thesis]. Simon Fraser University, 2004.

[52] Goldberg E, Novikov Y. BerkMin: A fast and robust SAT-solver. Proc. of Design, Automation, and Test in Europe Conference and Exposition (DATE). 2002. 131–149. [doi: 10.1016/j.dam.2006.10.007]

[53] Liang JH, Ganesh V, Poupart P, Czarnecki K. Exponential recency weighted average branching heuristic for SAT solvers. Proc. of the AAAI Conference on Artificial Intelligence. 2016. 3434–3440.

[54] Ansótegui C, Giráldez-Cru J, Levy J. The community structure of SAT formulas. International Conference on Theory and Applications of Satisfiability Testing. Springer. 2012. 410–423. [doi: 10.1007/978-3-642-31612-8_31]

[55] Cherif MS, Habet D, Terrioux C. Kissat MAB: Combining VSIDS and CHB through multi-armed bandit. SAT Competation 2021.15.

[56] Tseitin GS. On the complexity of derivation in propositional calculus. In: Siekmann J, Wrightson G, eds. Automation of Reasoning: 2: Classical Papers on Computational Logic. Springer, Berlin, Heidelberg, 1983: 466–483.

[57] Biere A. Deep bound hardware model checking instances, quadratic propagations benchmarks and reencoded factorization problems submitted to the SAT competition 2017. Proc. of SAT Competition. 2017. 40–41.

[58] Biere A. The AIGER and-inverter graph format [Technical report]. Johannes Kepler University. http://fmv.jku.at/aiger.

[59] Balyo T, Heule MJH, Järvisalo M. Preface. Proc. of SAT Competition 2017: Solver and Benchmark Descriptions. 2017.

[60] Bini D, Pan VY. Polynomial and Matrix Computations: Fundamental Algorithms. Birkhauser Verlag Basel, Switzerland, 2014.

[61] Zippel R. Effective Polynomial Computation. Springer Science & Business Media, 2012.

[62] Liang JH, Oh C, Ganesh V, *et al.* Maple-COMSPS, MapleCOMSPS LRB, MapleCOMSPS CHB. Proceedings of SAT Competition. 2016.

[63] Xiao F, Luo M, Li CM, *et al.* MapleLRB LCM, Maple LCM, Maple LCM dist, MapleLRB LCMoccrestart and Glucose-3.0+ width in SAT competition 2017. Proc. of SAT Competition. 2017. 22–23.

[64] Oh C. COMiniSatPS the Chandrasekhar Limit and GHackCOMSPS. SAT Competition. 2016.

[65] Eén N, Sörensson N. An extensible SAT-solver. International Conference on Theory and Applications of Satisfiability Testing. Springer. 2003. 502–518.

Mao Luo, Ph.D. candidate. His research interest is Boolean Satisfiability problem.

Chumin Li, Ph.D., professor. His research interests include the practical resolution of NP-hard problems, including SAT, CSP, MaxSAT, MinSAT, MaxClique, and GCP. He is particularly interested in the intrinsical relationships between these problems. One of his research directions is to find and exploit these relationships to solve them. A recent example is the exploitation of the relationships between MaxSAT and MaxClique to solve MaxClique.

Xinyun Wu, Ph.D., associate professor. His research focuses on implementing meta-heuristics on various NP-hard problems with graph structures, such as Traffic Grooming, RWA, Network Design, Dominating Set, etc.

Shuolin Li, Ph.D. candidate. Her research interest is Boolean Satisfiability problem.

Zhipeng Lü, Ph.D., professor. His research interests include artificial intelligence, computational intelligence, operations research and adaptive metaheuristics for solving large-scale real-world and theoretical combinatorial optimization, and constrained satisfaction problems.