More efficient stochastic local search for satisfiability

Huimin Fu, Guanfeng Wu*, Jun Liu and Yang Xu

Abstract

Uniform random satisfiability (URS) and hard random satisfiability (HRS) are two significant generalizations of random satisfiability (RS). Recently, great breakthroughs have been made on stochastic local search (SLS) algorithms for uniform RS, resulting in several state-of-the-art algorithms, e.g., Dimetheus, YalSAT, ProbSAT and Score$_2$SAT. However, compared to the great progress of SLS on URS, the performance of SLS on HRS lags far behind. In this paper, we propose two global clause weighting schemes and a new hybrid scoring function called *SA* based on a linear combination of a property *score* and property *age*, and then apply a second-level-biased random walk strategy based on two clause weighting schemes and *SA* to develop a new SLS solver called BRSAP. To evaluate the performance of BRSAP, we conduct extensive experiments to compare BRSAP with state-of-the-art SLS solvers and complete solvers on HRS instances and URS instances from SAT Competition 2017 and SAT Competition 2018 as well as 4100 generated satisfiable large HRS and URS ones. The experiments illustrate that BRSAP obviously outperforms its competitors, indicating the effectiveness of BRSAP. We also analyze the effectiveness of the underlying ideas in BRSAP.

Keywords: Hard random satisfiability (HRS) · Stochastic local search (SLS) ·Linear combination · Property

1 Introduction

The propositional satisfiability (SAT) problem is one of the most widely studied NP-complete problems and plays an outstanding role in many domains of computer science and artificial intelligence due to its significant importance in both theory and applications [1]. The SAT problem is fundamental in solving many practical problems in combinatorial optimization, statistical physics, circuit verification, computing theory [2, 14], and SAT algorithms have been widely used to solve real-world applications, such as computer algebra systems [9], core computer algebra systems [47], core graphs [48], gene regulatory networks [49], automated verification [54], model-based diagnosis (MBD)[55], scheduling [56], machine induction [57].

This work is supported by the National Natural Science Foundation of China (Grant No.61673320) and the Fundamental Research Funds for the Central Universities (Grant No.2682017ZT12, 2682016 CX119).

H. Fu is with the Key Laboratory of National-Local Joint Engineering Laboratory of System Credibility Automatic Verification of China, School of Information Science and Technology, Southwest Jiaotong University, Chengdu, China (email: fhm6688@my.swjtu.edu.cn)

*Corresponding author. Guangfeng Wu is with the Key Laboratory of National-Local Joint Engineering Laboratory of System Credibility Automatic Verification of China, School of Mathematic, Chengdu, China (email: wuguanfengfeng@126.com)

J. Liu is with the Key Laboratory of National-Local Joint Engineering Laboratory of System Credibility Automatic Verification of China, and also with the School of Computing, Ulster University, Northern Ireland, UK (email: j.liu@ulster.ac.uk)

Y. Xu is with the Key Laboratory of National-Local Joint Engineering Laboratory of System Credibility Automatic Verification of China, School of Mathematic, Chengdu, China (email: xuyang@swjtu.edu.cn)

There are many optimization algorithms dedicated to different SAT solvers to solving SAT problems, which are divided into two main classes: one is complete, the other is incomplete.

Complete algorithms are mainly based on Davis-Putnam-Logemann-Loveland algorithm (DPLL) [3, 4] and resolution principle [5]. DPLL algorithm is based on a binary search tree and adopts chronological backtracking, while the Conflict-Driven Clause Learning CDCL algorithm [58] maintains a stack of assumptions and propagations and adopts non-chronological backtracking as well as chronological backtracking [60]. The direct improvement on DPLL is to extend it into lookahead heuristics, which utilizes global heuristics to pick good decisions at the top-level [59].

The incomplete SAT solvers are mainly based on stochastic local search (SLS) algorithms [6, 7] which are among the best-known methods currently available for solving types of SAT problems. Although the incomplete SAT solvers cannot guarantee either to find the solutions or prove a given Boolean formula unsatisfiable, some of them are surprisingly more effective than state-of-the-art complete solvers on finding models of satisfiable formulae for random *k*-SAT instances [8]. The heuristics used by SLS solvers to solve random SAT problems are also potentially useful for solving real-world SAT problems [9].

In this work, we concentrate on the SLS algorithm. SLS algorithms are best suited for solving problems required short time to solve. [1]. There are more interests in improving the performance of SLS algorithms on random SAT instances, especially hard random SAT (HRS) ones [46]. From the theoretical viewpoint, HRS is a random 3-SAT, which is a classic problem in computational complexity research. From the practical viewpoint, in addition to being applied to sat solving, heuristic methods have also been applied to solve a variety of problems in the field of machine learning and artificial intelligence, and it still has great potential in application, e.g., generators of HRS with a predefined solution can be used in cryptography as one-way functions [10].

In the beginning, an SLS algorithm generally generates an initial assignment of the variables of *F*. Then it explores the search space to minimize the number of unsatisfied clauses. To do this, it iteratively flips the truth value of a variable selected according to some heuristic at each step until it seeks out a solution or timeout. Hence, there are two main factors affecting SLS algorithms, one is to generate a clause selection heuristic, and the other is a variable selection heuristic.

In focused random walk (FRW) algorithms, SLS solvers generally select a clause from unsatisfied clauses randomly, such as ProbSAT [17], YalSAT [20], Dimetheus [16], WalkSATlm [18]. Most SLS solvers improve different variable selection heuristics to develop algorithms, and they usually use *make* property, *break* property and *score* property to decrease the current number of unsatisfied clauses, and utilize *age* property to avoid local optima.

In two-mode SLS algorithms during the last ten decades, the most significant development was perhaps "configuration checking" strategy (CC) [39] and "weights" strategy [19] (similar to "score function" [30]), leading to the effective CCASat [39], Swqcc [61], CScoreSAT [30] and DCCASat [19]. One of the main features of the CC strategy is that the last flipping variable must not be the current flipping variable [39]. One of the main features of the weighting schemes is that greedily select a best variable to be flipped among the candidate variables.

There have been numerous works on improving the performance of SLS algorithms [11-17]. Substantial progress has been made in only URS instances with various clause-to-variable ratios. However, a family of SAT instances includes URS instances and HRS instances, and most SLS algorithms on random instances focus on URS. Although URS at the phase transition has been cited as the hardest track of SAT problems [18, 19], when it comes to the HRS instances, which is even harder than URS instances at the solubility phase transition for SLS solvers, Dimetheus[13] , ProbSAT[17], Yalsat [20] and Score$_2$SAT [22] lost their power and effectiveness, as can be seen from the competition results of the random track of SAT Competition 2017[1] and 2018[2], so their performance for solving HRS need be further improved. Compared to the great progress of SLS algorithms on solving URS, the performance of SLS algorithms on solving HRS lags far behind. This motivates us to design a more efficient SLS algorithm for solving HRS.

This paper is devoted to developing an efficient SLS algorithm for solving HRS and URS instances. The improvement of weighting schemes has become the mainstream of optimizing SLS algorithms [51-53]. In this work, we propose two ideas about clause weighting schemes. The first and most important one is based on an intuition that prefers to satisfy frequently becoming unsatisfied or easily keeping satisfied clauses during the search. This is done by two new clause weighting schemes that work for unsatisfied clauses in the total search and is activated to pick a clause. It is worth noting that previous SLS algorithm for SAT either do not use clause weighting scheme or update clause' weights when a local optimum is reached and utilize the clause weighting scheme is to select a variable. Our work develops a second-level-biased random walk based on two global clause weighting schemes to select a clause. We also propose a new scoring function named *SA* based on a linear combination of *score* property and *age* property. The *SA* function differs from previous hybrid scoring functions in that it considers one level *score* property distinguishing itself from previous two levels *score* property in SLS algorithms [30]. Based on *SA*, we design a new tie-breaking strategy. Then based on the second-level-**b**iased **r**andom **w**alk and the scoring function ***SA***, we develop a new SLS algorithm called BRSAP (combining second-level-**b**iased **r**andom walk based on two new clause weighting schemes and linear scoring function ***SA*** as well as the **p**robability strategy). To evaluate the effectiveness of the BRSAP algorithm, we conduct extensive experiments on HRS instances to compare BRSAP against recent state-of-the-art SLS algorithfms, including CSoreSAT [30], Score$_2$SAT [22], YalSAT, Sparrow [23], ProbSAT and Dimetheus, and state-of-the-art two complete algorithms SparrowToRiss [23] and gluhack [24] on HRS instances. The solvers are compared on HRS and URS problems from the SAT Competitions in 2017 and 2018 and on randomly generated HRS and URS problems. The experimental results show that BRSAP performs remarkably well compared to state-of-the-art algorithms on HRS instances. BRSAP also proves to be competitive even when it is compared to state-of-the-art algorithms like ProbSAT, YalSAT, CscoreSAT, Score$_2$SAT and SparrowToRiss on URS with long clauses. Moreover, through the analysis on the experimental results, it has proved performance superiority of the underlying ideas in BRSAP.

This paper is organized as follows. In Section 2, we provide some necessary basic knowledge. Section 3 reviews the definition of polynomial probability. In Section 4, we introduce two clause weighting schemes. Section 5 provides the biased random walk. In Section 6, we present the new tie-breaking based on the new scoring function *SA* and the BRSAP algorithm in detail. The experimental analyses and some discussions are performed in Section 7 and Section 8, respectively. Finally, we conclude this paper and then give some future work in Section 9.

2 Preliminaries

A formula *F* of the SAT is defined by a pair *F*=(*X*, *C*) such that $X=\{x_1, x_2,..., x_n\}$ is a set of *n* Boolean variables (their values belong to the set {true, false}) and $C=\{c_1, c_2, ..., c_n\}$ is a set of *m* clauses. A clause $c_i \in C$ is a disjunction of literals and a literal is either a variable x_i (which is called positive literal) or its

[1] https://baldur.iti.kit.edu/sat-competition-2017/results/random.csv.
[2] http://sat2018.forsyte.tuwien.ac.at/index.php?cat=results.

negation $\neg x_i$ (which is called negative literal). If the size of each clause in C is equal to k, then the instance is a k−SAT instance and $r= m/n$ is its clause-to-variable ratio. An instance $F= c_1 \wedge c_2 \wedge ... \wedge c_m$ is a conjunction of clauses.

A complete satisfying assignment α for a formula F is an assignment to its variables making formula F true. If x_i is true by α then x_i belongs to α (otherwise $\neg x_i \in \alpha$). A literal l is said to be satisfied by the current value of the variable α if $l \in \alpha$ and falsified if $\neg l \in \alpha$. A clause is unsatisfied by α if its all literals are false literal and satisfied otherwise. A satisfying solution of F is a complete assignment that satisfies all the clauses of F.

In SLS algorithms for HRS, for a variable x and an assignment α, the mainly variable x properties used by SLS algorithms for SAT are *make*(x) [25] and ***break***(x) [26], which are the number of clauses that would become satisfied and unsatisfied respectively, if variables x were to be flipped. Usually, SLS algorithms for random k-SAT instances select a variable x to be flipped based on its properties of *score*(x) [27-30] and ***age***(x) [31-35]. A scoring function which can be a simple property or any mathematical expression with one or more properties measures the increase in the number of satisfied clauses by flipping x, and ***score***(x) is defined as *make*(x)–*break*(x). *age*(x) is defined as the number of steps that have occurred since the variable x was last flipped [30].

The hard random SAT (HRS) is particularly interesting because it turns out to be one of the hardest for all solvers [10]. Moreover, the HRS instances generated are especially difficult for SLS algorithms [36]. Parameter optimization tool SMAC [37], has been successful in improving the performance of SAT solvers, especially SLS solvers. However, the recent successful generator is based on the clause distribution control method [38] and SMAC with the opposite purpose to slow down SAT solvers and can be automatically configured to generate hard benchmarks based on Dimetheus, ProbSAT and so on [10].

HRS was added for the first time to the random track of SAT Competition in 2016 in order to evaluate and improve SAT solvers, especially for SLS solvers. As witnessed in SAT competitions since 2016, it has become a mainstream for SLS solvers, for example, apart from URS instances, most (nearly 65% of) instances in the benchmark of the random SAT track in the SAT Competition 2018 are HRS, which are classified into three types based on clause-to-variable ratios (r): r=4.3, r=5.206 and r=5.5. However, the performance of existing SLS algorithms lags far behind on HRS especially for ratios of r=5.206 and 5.5.

Clause selection heuristic and variable selection heuristic are two main factors in affecting SLS algorithms. In order to develop SLS algorithms for HRS, we focus on clause and variable selection.

3 Reviewing probability strategy

In this section, we briefly review the probability strategy in ProbSAT [26].

Probability strategy has presented success on applying in SLS algorithms. In the context of SAT, the first definition of probability strategy based on the combination of *break* and *make* has been introduced in the literature [26]. An alternative notion of probability strategy [17, 26] base on only *break* has been proposed, and in the literature [26] probability strategy based *break* has shown the superiority on solving SAT problem. The probability strategy based only *break* leads several SLS algorithms for SAT [13, 15, 17, 20, 26]. In this work, we adopt the definition of probability strategy based only *break* [17].

The probability strategy called $f(x, \alpha)$ [26] including a polynomial or exponential uses only the *break* values of a variable x under a complete assignment a as listed below.

$$f(x, \alpha)=(0.9+break(x, \alpha))^{-2.06}$$

$$f(x, \alpha)=(c_b)^{-break(x,a)}$$

According to the criterion of the algorithm based on probability strategy $f(x, \alpha)$, given a selected clause c, the algorithms utilize $\frac{f(x,\alpha)}{\sum_{z\in C} f(z,\alpha)}$ to probabilistically select variables that have smaller *break* values [17].

4 Two clause weighting schemes

In this section, we introduce two new clause weighting schemes in the total search. Based on these clause weighting schemes, we define some new types of clauses.

Clause weighting schemes have been used prominently in SLS algorithms for solving SAT [22, 30, 31], such as SWT [39], DLM [40], PAWS [41], SAPS [42]. Although these clause weighting SLS algorithms differ in the manner clause' weights should be updated (probabilistic or deterministic), they all choose to increase the weights of all the unsatisfied clauses or reduce the weights of all the satisfied clauses as soon as a local minimum is encountered. Recent studies, mainly including CCASat [39] CSCCSat [21], Score$_2$SAT as well as their variant considered that the algorithm should be forced to satisfy more clauses, and the weights of clauses should be updated when the search is stuck in a part [39, 43]. These clause weighting techniques turn out to be essentially ineffective for solving HRS instances.

But better weighting techniques can be derived by taking a global scheme. It happens that the algorithm without using the clause weighting scheme has loss some clauses that are difficult to satisfy before it gets stuck in a "stuck" state. Therefore, forcing the algorithm to satisfy more clauses will mislead the algorithm to obtain worse quality allocation. To avoid this situation happening, we consider two global schemes named GWU and GWAC that update the clause' weights in the total search process respectively.

4.1 The clause weighting scheme GWU

The first clause weighting scheme is denoted by GWU (**G**lobal **W**eight based on **U**nsatisfied clauses) and works as follows. For each clause c in step s, we associate an integer number $GWU(c, s)$ as its weight. **Whenever a variable is selected to be flipped**, then clause' weights are updated as follow:

- In the beginning of the SLS algorithm, for each clause *c*, if *c* is unsatisfied under the initial assignment *a*, *c*'s weight is initialized to 1 (i.e., *GWU* (*c*, 0) =1); otherwise, *c*'s weight is initialized to 0 (i.e., *GWU* (*c*, 0) =0).
- When SLS algorithm searches to step *s*, and if a clause *c* is unsatisfied, the clause *c*'s weight is activated, then *GWU* (*c*, *s*) is equal to *GWU* (*c*, *s*-1) + 1; otherwise, *GWU* (*c*, *s*) is equal to *GWU* (*c*, *s*-1).

Thus, in the implementation of the proposed GWU scheme, a clause *c*'s weight has been changed if *c* is unsatisfied whenever a variable is picked to be flipped.

Although there is similar idea between GWU and SWT [39], they have important differences. If the clause is unsatisfied, then the clause' weights based GWU is initialized as 1, otherwise, clause' weights is initialized to 0, while all clauses' weights based on SWT are initialized as 1. Moreover, whenever a variable is selected to be flipped, then GWU is called, i.e., the weights of all unsatisfied clauses increased by one, while when the algorithm falls into local optimum, the SWT is only called, i.e., the weights of all unsatisfied clauses increased by one, but if the average weight exceeds a threshold, it needs to smooth all clause' weights.

Previous algorithms select a clause from the unsatisfied clauses with equal probability [13,16, 17, 20, 26] i.e., simply categorizing clauses into unsatisfied ones and satisfied ones is not informative enough to guide the SLS, especially for HRS instances.

Thus, suggested by "deceasing" variables (comprehensively decreasing variables) variables [30] in SLS algorithms, we develop two sets of HSC-GWU (**h**ard **s**atisfiable **c**lauses based on **GWU**) and ESC-GWU (**e**asily **s**atisfiable **c**lauses based on **GWU**) to distinguish unsatisfied clauses. The formal definitions of HSC-GWU and ESC-GWU are given as follows:

Definition 1. *For a CNF formula F, a positive integer parameter β, when SLS algorithm runs to step s, a clause c is a HSC-GWU in step s if and only if c is unsatisfied and GWU*(*c*, *s*)/100$\geqslant\beta$.

Definition 2. *For a CNF formula F, a positive integer parameter β, when SLS algorithm runs to step s, a clause c is a ESC- GWU in step s if and only if c is unsatisfied and GWU* (*c*, *s*)/100$<\beta$.

Note that the purpose of the *GWU* (*c*, *s*) modules 100 is to prevent the setting of positive integer parameter β from being too large.

In this work, when SLS algorithm searches to any step *s*, we use the notation *HSC-GWU*(*s*) to denote the set of all HSCs-GWU in step *s* and *ESC-GWU*(*s*) to denote the set of all ESCs-GWU in step *s*. In the step *s*, the union of *HSC-GWU*(*s*) and *ESC-GWU*(*s*) is the set of all unsatisfied clauses at step *s*.

The intuition that clauses with larger *GWU* values are harder to keep satisfied in the search process. Thus, it is beneficial for SLS algorithms to prefer satisfying HSCs-GWU, and we use GWU to guide clause selection.

HSCs-GWU are regarded as the good candidates of clauses to be selected in the clause selection heuristic for solving SAT, that means HSCs-GWU are put higher priority to be satisfied in each search step.

Based on the notions of HSCs-GWU and ESCs-GWU, until at least β*100 steps, all the unsatisfied clauses are ESCs-GWU in each step, and then the same problem is that the algorithm cannot distinguish the unsatisfied clauses in the clause selection. Thus, this motivates us to design the second new clause weighting scheme which could distinguish ESCs-GWU.

4.2 The clause weighting scheme GWAC

As the *age* property of variables is diversification mode, which may be able to better handle local minimum. We propose a new clause weighting scheme based on the *age* property of clauses.

The second clause weighting scheme is denoted by GWAC (**G**lobal **W**eight based on **A**ge property of **C**lause) and works as follows. For each clause *c* in step *s*, we associate an integer number *GWAC* (*c*, *s*) as its weight. **Whenever a clause is selected by heuristics**, then clause' weights are updated as follow:

- In the beginning of the SLS algorithm, for each clause *c*, *c*'s weight is initialized to 0 (i.e., *GWAC* (*c*, 0) =0).
- When SLS algorithm searches to step *s*, *GWAC* (*c*, *s*) is the number of steps that have occurred since the clause *c* was last selected.

Thus, in the implementation of the proposed GWAC scheme, a clause *c*'s weight has been changed in each step.

Although there is similar idea between *age* and GWAC, they are an important difference. The GWAC is adjusted for clause, while the *age* property [30] is for variable.

Based on the GWAC, we also develop two sets of LAC-GWAC (**l**ong **a**ge **c**lause based on **GWAC**) and SAC-GWAC (**s**hort **a**ge **c**lause based on **GWAC**) to distinguish ESCs-GWU. The formal definitions of LAC -GWAC and SAC-GWAC are given as follows:

Definition 3. *For a CNF formula F, a positive integer parameter η, when SLS algorithm runs to step s, a clause c is a LAC-GWAC in step s if and only if c is* ESC-GWU *and GWAC*(*c, s*)$\geqslant\eta$.

Definition 4. *For a CNF formula F, a positive integer parameter η, when SLS algorithm runs to step s, a clause c is a SAC- GWAC in step s if and only if c is* ESC-GWU *and GWAC* (*c, s*) $<\eta$.

Note that the parameter η is positive integer.

In this work, when SLS algorithm searches to any step *s*, we use the notation *LAC- GWAC*(*s*) to denote the set of all LACs-GWAC in step *s* and *SAC-GWAC*(*s*) to denote the set of all SACs-GWAC in step *s*. In the step *s*, the union of *LAC- GWAC* (*s*) and *SAC-GWAC*(*s*) is the set of ESCs-GWU at step *s*.

The intuition that clauses with larger GWAC values are easier to keep satisfied in the search process, and GWAC is a supplement to GWU. If the algorithm only depends on GWU to pick a clause, it will easily fall into local optimization. Thus, if there is no HSCs-GWU, it is beneficial for SLS algorithms to select a LAC-GWAC. We use GWAC to guide clause selection.

If there is no HSCs-GWU, LACs-GWAC are regarded as the good candidates of clauses to be selected in the clause selection heuristic for solving SAT, that means LACs-GWAC are put the second higher priority to be satisfied in each search step.

Here we utilize the GWU and GWAC for picking a clause, distinguishing itself from previous clause weighting schemes in SLS algorithms for picking a variable [15,17,19,21,22, 39].

5 Second-level-biased random walk based on GWU and GWAC

The random walk strategy is a standard component designed for SAT. However, the standard random walk strategy may not be suitable for SLS algorithms for HRS, because it does not distinguish between HSCs-GWU and ESCs-GWU, or between LACs-GWAC and SACs-GWAC. Since HSCs-GWU and LACs-GWAC are put higher priority to be selected for SAT in the proposed algorithm, thus it is reasonable for us to develop a second-level-biased random walk component. The second-level-biased random walk strategy is suggested by the idea from [45] and described as follows:

- When the second-level-biased random walk is called, if there exists HSCs-GWU, the algorithm selects an HSC-GWU randomly;
- Otherwise, if there exists LACs-GWAC, the algorithm selects an LAC-GWAC randomly;
- If there is no LACs-GWAC, the algorithm picks an ESC-GWU or SAC-GWAC randomly;
- Then, the algorithm picks a variable to be flipped in the chosen clause. In this work, this is accomplished by a variable selection strategy which is described in the subsequent section.

Although there is similar idea between second-level-biased random walk strategy and biased random walk strategy, they are an important difference. The second-level- biased random walk strategy is utilized to select a clause from two higher priority of sets, while the biased random walk strategy is used to select a clause for one higher priority of set.

By combining HSCs-GWU harder to keep satisfied and LACs-GWAC easier to keep satisfied, the second-level-biased random walk can maintain a balance between intensification and diversification, making the SLS algorithm more widely applicable.

6 The scoring function *SA* and BRSAP algorithm

In this section, we first propose a new scoring function named *SA* which combines a *score* (greedy property) and an *age* (diversification property) in a linear combination, and then we utilize the SA to develop a new tie-breaking strategy.

6.1 The scoring function *SA*

Heuristics in SLS algorithms for SAT mainly include two-mode SLS algorithms [1, 21, 22, 23, 30] and focused random walk (FRW) algorithms [16, 17, 18, 20, 25, 26]. FRW algorithms always select a variable to be flipped from an unsatisfied clause chosen randomly in each step [7]. Based on Section 4 and Section 5, our algorithm belongs to FRW algorithms.

For SLS algorithms, there is one important issue that is tie-breaking –In SLS algorithm, tie-breaking strategy makes the algorithm select a variable to flip when faced with multiple candidate variables.) [1, 7, 50]. However, in FRW algorithms, there is still other important issue - that generally may result in the same variable being selected in consecutive steps (we also call this issue tie-breaking). Actually, there is almost no previous work devoted to handling this problem for FRW algorithms. To avoid this, inspired by the previous tie-breaking in Ref. [1, 30], we employ a new tie-breaking based on a new scoring function named *SA* combining greedy property *score* and diversification property *age*. The definition of *SA* is given below.

Definition 5 *Given a CNF formula F, for a variable x, in search step s, when the assignment is α, the scoring function, denoted as SA, is defined as:*

$$SA\,(x, s, \alpha) = score\,(x, \alpha) + age\,(x, s)/\mu,$$

where μ is a positive integer parameter, which is used to control the role of the *age* value played in the scoring function.

The new tie-breaking based on a linear scoring function *SA* can also maintain a balance intensification and diversification.

6.2 The BRSAP algorithm

In this subsection, we utilize the second-level-**b**iased **r**andom walk based on two new clause weighting schemes and linear scoring function ***SA*** as well as the **p**robability strategy to develop a new SLS algorithm called BRSAP.

The pseudo-code of the BRSAP algorithm is outlined in Algorithm 1 and it can be described in detail as follows.

At the start of the algorithm, BRSAP performs the first loop until it finds a satisfying assignment or reaches the first limited steps denoted by *MaxSteps* (line 2 in Algorithm 1). Then BRSAP generates a complete assignment α randomly as the initial solution (line 3). *bestVar* is used to record which variable was flipped in the last step (line 4). Then we initialize *GWU*(*c*,0) and *GWAC*(*c*,0) as 0 for each clause *c* as well as *HSC-GWU* (0), *ESC-GWU* (0), *LAC-GWAC* (0) and *SAC- GWAC* (0) as 0 (line 5 in Algorithm 1).

After the initialization, BRSAP executes the second loop until a satisfying solution is found or exceeds the second limited steps *MaxTries* (line 7). In each search step, BRSAP selects a variable to be flipped.

Firstly, BRSAP picks a clause based on the second-level-

Agorithm 1: BRSAP algorithm

```
   Input: CNF-formula $F$, $MaxTries$, $MaxSteps$, $\mu$, $\beta$, $\eta$
   Output: A satisfying assignment $\alpha$ of $F$, or "UNKNOWN"
1  begin
2      for $i$ = 1 to $MaxTries$ do
3          $\alpha$ :=a generated truth assignment randomly for $F$;
4          $bestVar$ :=null;
5          Initialize $GWU(c,0)$ and $GWAC(c,0)$ for each clause $c$ and $HSC\text{-}GWU(0)$, $ESC\text{-}GWU(0)$, $LAC\text{-}GWAC(0)$ and $SAC\text{-}GWAC(0)$ as 0.
6          compute $score(x, a)$;
7          for $j$ = 1 to $MaxSteps$ do
8              if $\alpha$ satisfies $F$ then Return $\alpha$;
9              if $HSC\text{-}GWU(j)$ is not empty then
10                 $C$ := a clause randomly chosen from $HSC\text{-}GWU(j)$;
11             else
12                 if LAC-GWAC($j$) is not empty then
13                     $C$ := a clause randomly chosen from $LAC\text{-}GWAC(j)$;
14                 else
15                     $C$ := a clause randomly chosen from $SAC\text{-}GWAC(j)$;
16             update GWAC;
17             $v := x \in C$ selected according to probability $\frac{f(x,\alpha)}{\sum_{z\in C} f(z,\alpha)}$;
18             If $v$ :==$bestVar$ then
19                 $bestVar := x \in C, x \neq v$, with the greatest $SA(x, j, \alpha)$;
20             else
21                 $bestVar := v$;
22             $\alpha$:= $\alpha$ with $bestVar$ flipped;
23             update GWU and $age(x, j)$ for each variable $x$;
24     Return "UNKNOWN";
25 end
```

biased random walk strategy as detailed in Section 5. If HSC-GWU (*j*) is not empty in any step *j*, a clause is picked randomly from HSC-GWU (*j*) (lines 9 and 10); otherwise, if LAC-GWAC (*j*) is not empty in any step *j*, a clause is picked randomly from LAC-GWAC (*j*)((lines 11-13), and if the LAC-GWAC (*j*) is empty, a clause is picked randomly from SAC-GWAC (*j*) (or ESC-GWU (*j*)) (lines 14 and 15), and then updates the clause' weights based on the weighting scheme GWAC detailed Section 4.2 (line 16).

Then BRSAP tries to pick a variable to be flipped according to the probability based on *f* and the new tie-breaking strategy as detailed in Section 6.1 (lines 17-21 in Algorithm 1): BRSAP first picks a variable by the probability based on *f* (if *k*=3, *f* uses polynomial strategy, otherwise, *f* uses exponential strategy) (line 17 in Algorithm 1), and then if the variable is the same as the last flipped variable (line 18), BRSAP picks a variable by preferring the variable with the greatest *SA* value (lines 19). After the variable is selected, the BRSAP flips the selected variable (line 22) and then updates the clause' weights based on the weighting scheme GWU detailed Section 4.1 (line 23), then the BRSAP algorithm starts the next search step.

Finally, once the search process terminates, the BRSAP reports α as the solution; otherwise, BRSAP reports UNKNOWN (line 24).

7 Experimental evaluation

In this section, in order to present the effectiveness of the BRSAP algorithm, we conduct extensive experiments to evaluate BRSAP on HRS and URS instances, and compare BRSAP against six state-of-the-art SLS solvers including CSoreSAT, Score$_2$SAT, YalSAT, ProbSAT, Sparrow and Dimetheus as well as two state-of-the-art complete solvers SparrowToRiss and gluhack on the same instances.

We first introduce the benchmarks, the competitors and experimental preliminaries. Then we compare BRSAP with state-of-the-art SLS solvers and complete solvers on all testing HRS and URS benchmarks.

7.1 Experimental evaluation on HRS

7.1.1 The HRS benchmarks

All the HRS instances used in our experiments are generated according to the HRS tool [10]. We adopt the following seven testing benchmarks.

1) **4.3HRS SAT2017**: all HRS instances with *r*=4.3 from SAT Competition 2017 [3] (*n*=400, 420, …, 540, 40 instances, 5 for each size)
2) **4.3HRS Random**: HRS instances generated randomly by the HRS tool (*r*=4.3, *n*=600, 700, …,1000, 1000 instances, 200 for each size)
3) **5.206HRS SAT2017**: all HRS instances with *r*=5.206 from SAT Competition 2017 (*n*=400, 420, …, 540, 40 instances, 5 for each size)
4) **5.206HRS Random**: HRS instances generated randomly by the HRS tool (*r*=5.206, *n*= 600, 700, …,1000, 1000 instances, 200 for each size)
5) **5.5 HRS SAT2017**: all HRS instances with *r*=5.5 from SAT Competition 2017 (*n*=400, 420, …, 540, 40 instances, 5 for each size)
6) **5.5HRS Random**: HRS instances generated randomly by the HRS tool (*r*=5.5, *n*= 600, 700, …,1000, 1000 instances, 200 for each size)
7) **5.699HRS Random**: HRS instances generated randomly by the HRS tool (*r*=5.699, *n*=200, 300, …,1000, 900 instances, 100 for each size)

7.1.2 The competitors

We compare the BRSAP algorithms with six state-of-the-art SLS solvers including CSoreSAT [30], Score$_2$SAT [22], YalSAT [20], ProbSAT [17], Sparrow [23] and Dimetheus [16] as well as two state-of-the-art complete solvers SparrowToRiss [23] and gluhack [24] on the same instances.

The CSoreSAT solver utilizes two scoring functions and

[3]https://baldur.iti.kit.edu/sat-competition-2017/benchmarks/

clause weighting scheme PAWS [41]. The Score$_2$SAT adopts two scoring functions and two clause weighting schemes SWT [19] and PAWS. Score$_2$SAT is the third place in SAT Competition 2017. YalSAT wins the random track of SAT Competition 2017. ProbSAT wins the random track of SAT Competition 2013, and is the second place among the SLS algorithms in SAT Competition 2018. Dimetheus is the winner of random SAT track of SAT Competition 2014 and SAT Competition 2016 respectively, and is the first place among the SLS algorithms in random SAT track of SAT Competition 2018. Sparrow uses the clause weighting scheme PAWS, and is the first place in the random SAT track of SAT Competition 2011. SparrowToRiss is a combination of Sparrow and Riss [23], and is the first place on the random SAT track of SAT Competition 2018. The gluHack is an efficient complete solver and wins the silver of SAT Competition 2018.

7.1.3 Experiment preliminaries

The BRSAP algorithm is implemented in C/C++. The BRSAP algorithm is involved in three parameters, i.e., β controlling the number of HSCs-GWU, η controlling the number of LACs-GWAC, γ controlling the balance between the *score* and *age*.

We tuned the β, η and γ parameters of BRSAP on HRS according to our experience in Table 1. For c_b, we utilize the default parameter setting tuned in the literature [17].

Table 1: Parameter settings of BRSAP for HRS instances

	μ	r=4.3	r=5.206/5.5	r=5.699
		η, β=9	β, η=312	β, η=312
$n\leq400$	900	321	800	500
$400<n<600$		276	1081	
$n\geq600$			1255	661

The binary of CScoreSAT is provide by its author. For the YalSAT and Score$_2$SAT solvers, we adopt the two codes submitted to SAT Competition 2017[4]. The binaries of ProbSAT, Dimetheus, Sparrow, SparrowToRiss and gluhack can be downloaded online[5] and we use the parameter setting as the one used in SAT Competition 2018.

Experiments on the seven benchmarks are carried out on Intel(R) Core (TM) i5-8265U 1.60 1.80GHz CPU with 8GB RAM, running the 64-bit Ubuntu Linux operating system. Each run that terminates in finding a satisfying assignment within the cutoff time is a successful run. The cutoff time is set to 600 seconds (as in the literature [36]) for 4.3HRS random benchmark, 5.206 HRS random benchmark, 5.5HRS random benchmark and 5.699HRS random benchmark, and 5000 seconds (as in SAT Competitions 2017 and 2018) for the rest benchmarks. For all benchmarks, each solver is executed 10 times for each instance. In this paper, for each solver on each instance group, we report the number of success runs (#suc) for the top seven benchmarks as well as "par 2", which is a penalized average run time where an unsuccessful run of a solver is penalized as 2 times cutoff time, and "Overall" symbols averaged over all instances with each run per instance. Note that PAR 2 is adopted in SAT Competitions and has been widely used in the literature [30]. The best results for an instance class are highlighted in **bold**. If a solver has no successful run on an instance class, the corresponding "par 2" is marked with "-".

7.1.4 Experimental results

In this subsection, we conduct extensive experiments of BRSAP and its state-of-the-art SLS and state-of-the-art complete competitors on all testing benchmarks, i.e., the 4.3HRS SAT17, 4.3HRS Random, 5.206HRS SAT17, 5.206HRS Random, 5.5HRS SAT17, 5.5HRS Random, and 5.699HRS Random.

7.1.4.1 Results on the 4.3HRS Random benchmark

Table 2 presents the comparative performance results of BRSAP and its state-of-the-art SLS competitors CSoreSAT, Score$_2$SAT, YalSAT, ProbSAT, Dimetheus as well as Sparrow and complete competitors gluHack and SparrowToRiss on the HRS instances with r=4.3 from SAT Competition 2017. On the overview of the results, BRSAP provides a better performance than gluHack and SparrowToRiss in terms of metrics. Overall, although BRSAP is slower than Score$_2$SAT, YalSAT, ProbSAT and Sparrow in terms of par 2, BRSAP and its competitors solve the same number of instances, indicating BRSAP is competitive with state-of-the-art SLS solvers, i.e., CSoreSAT, Score$_2$SAT, YalSAT, ProbSAT, Dimetheus and Sparrow.

7.1.4.2 Results on the 4.3HRS Random benchmark

Table 3 reports the comparative performance results of BRSAP and its state-of-the-art SLS competitors including CSoreSAT, Score$_2$SAT, YalSAT, ProbSAT, Dimetheus as well as Sparrow and complete competitors containing gluHack as well as SparrowToRiss. According to Table 3, BRSAP significantly outperforms its complete competitors gluHack and SparrowToRiss in terms of metrics. Although BRSAP performances slightly worse than CSoreSAT, Score$_2$SAT, YalSAT, ProbSAT, Dimetheus and Sparrow in terms of par 2, BRSAP and its SLS competitors show the same performance in terms of successful runs. Overall, BRSAP outperforms SparrowToRiss in terms of par 2.

7.1.4.3 Results on the 5.206HRS SAT2017 benchmark

The comparative results of BRSAP and its state-of-the-art SLS competitors CSoreSAT, Score$_2$SAT, YalSAT, ProbSAT, Dimetheus as well as Sparrow, and complete competitors gluHack as well as SparrowToRiss on the HRS instances with r=5.206 from SAT Competition 2017 are summarized in Table 4. Overall, BRSAP and SparrowToRiss succeed in all runs, while Score$_2$SAT, Sparrow and gluHack only succeed in 80, 80 and 380 runs (out of 400 runs) respectively, and also CSoreSAT, YalSAT, ProbSAT and Dimetheus fail to solve any instance on this benchmark. According to the empirical results presented in Table 4, BRSAP solves each instance within one second. More

[4]https://baldur.iti.kit.edu/sat-competition-2017/index.php?cat=downloads
[5]http://sat2018.forsyte.tuwien.ac.at/index.php?cat=downloads

Table 2: **Experimental results on the 4.3HRS SAT2017 benchmark**.

Instance Class	CSoreSAT	Score$_2$SAT	YalSAT	ProbSAT	Dimetheus	Sparrow	gluHack	SparrowToRiss	BRSAP
	#suc	#suc	#suc	#suc	#suc	#suc	#suc	#suc	#suc
	par2	par2	par2	par2	par2	par2	par2	par2	par2
n=400	50	50	50	50	50	50	50	50	50
	0.006	**0.000**	0.010	0.004	0.014	0.002	328.5	0.068	0.016
n=420	50	50	50	50	50	50	50	50	50
	0.018	**0.002**	0.008	**0.002**	0.028	**0.002**	619.1	0.059	0.014
n=440	50	50	50	50	50	50	30	50	50
	1.766	**0.022**	0.072	0.004	0.331	0.192	4702	0.338	0.052
n=460	50	50	50	50	50	50	50	50	50
	0.022	0.010	0.014	0.006	0.009	**0.002**	1521	0.058	0.044
n=480	50	50	50	50	50	50	40	50	50
	0.000	0.008	0.002	**0.000**	0.011	**0.000**	2410	0.067	0.024
n=500	50	50	50	50	50	50	20	50	50
	0.002	**0.000**	**0.000**	**0.000**	0.009	**0.000**	6461	0.060	0.012
n=520	50	50	50	50	50	50	30	50	50
	0.004	0.006	0.008	**0.000**	0.010	**0.000**	4891	0.072	0.026
n=540	50	50	50	50	50	50	20	50	50
	0.004	0.010	0.004	**0.000**	0.010	**0.000**	6910	0.075	0.022
Over all	400	400	400	400	400	400	290	400	400
	0.228	0.007	0.015	**0.002**	0.053	0.025	3480	0.100	0.026

Table 3: **Experimental results on the 4.3HRS Random benchmark**.

Instance Class	CSoreSAT	Score$_2$SAT	YalSAT	ProbSAT	Dimetheus	Sparrow	gluHack	SparrowToRiss	BRSAP
	#suc	#suc	#suc	#suc	#suc	#suc	#suc	#suc	#suc
	par2	par2	par2	par2	par2	par2	par2	par2	par2
n=600	2000	2000	2000	2000	2000	2000	400	2000	2000
	0.006	0.006	0.004	**0.000**	0.010	0.002	979.9	0.063	0.026
n=700	2000	2000	2000	2000	2000	2000	0	2000	2000
	0.012	0.022	0.024	0.012	0.037	**0.004**	-	0.177	0.050
n=800	2000	2000	2000	2000	2000	2000	0	2000	2000
	0.026	0.056	0.018	0.032	0.059	**0.002**	-	0.394	0.320
n=900	2000	2000	2000	2000	2000	2000	0	2000	2000
	0.084	0.046	0.030	0.080	0.024	**0.006**	-	0.159	0.050
n=1000	2000	2000	2000	2000	2000	2000	0	2000	2000
	0.030	0.018	0.010	0.002	0.024	**0.000**	-	0.187	0.064
Over all	10000	10000	10000	10000	10000	10000	400	10000	10000
	0.032	0.030	0.017	0.025	0.031	**0.003**	1156	0.196	0.102

Table 4: **Experimental results on the 5.206HRS SAT2017 benchmark**.

Instance Class	CSoreSAT	Score$_2$SAT	YalSAT	ProbSAT	Dimetheus	Sparrow	gluHack	SparrowToRiss	BRSAP
	#suc	#suc	#suc	#suc	#suc	#suc	#suc	#suc	#suc
	par2	par2	par2	par2	par2	par2	par2	par2	par2
n=400	0	0	0	0	0	0	50	50	50
	-	-	-	-	-	-	45.30	2.102	**0.258**
n=420	0	40	0	0	0	30	50	50	50
	-	2000	-	-	-	4008	19.43	0.988	**0.590**
n=440	0	10	0	0	0	30	50	50	50
	-	8000	-	-	-	4007	44.65	0.652	**0.648**
n=460	0	0	0	0	0	20	50	50	50
	-	-	-	-	-	6002	238.0	0.982	**0.810**
n=480	0	10	0	0	0	0	50	50	50
	-	8000	-	-	-	-	504.0	14.65	**0.698**
n=500	0	0	0	0	0	0	50	50	50
	-	-	-	-	-	-	284.1	1.498	**0.762**
n=520	0	10	0	0	0	0	40	50	50
	-	8000	-	-	-	-	2686	17.97	**0.766**
n=540	0	10	0	0	0	0	40	50	50
	-	8000	-	-	-	-	3183	9.179	**0.836**
Over all	0	80	0	0	0	80	380	400	400
	-	8000	-	-	-	8002	875.6	6.003	**0.671**

Table 5: **Experimental results on the 5.206HRS Random benchmark**.

Instance Class	CSoreSAT	Score$_2$SAT	YalSAT	ProbSAT	Dimetheus	Sparrow	gluHack	SparrowToRiss	BRSAP
	#suc	#suc	#suc	#suc	#suc	#suc	#suc	#suc	#suc

	par2	par2	par2	par2	par2	par2	par2	par2	par2
n=600	0	0	0	0	0	400	400	2000	2000
	-	-	-	-	-	965.1	968.0	11.94	**0.890**
n=700	0	0	0	0	0	0	0	2000	2000
	-	-	-	-	-	-	-	12.11	**1.140**
n=800	0	0	0	0	0	0	0	1600	**2000**
	-	-	-	-	-	-	-	274.4	**1.388**
n=900	0	0	0	0	0	0	0	1200	**2000**
	-	-	-	-	-	-	-	505.6	**1.410**
n=1000	0	0	0	0	0	0	0	1200	**2000**
	-	-	-	-	-	-	-	504.9	**1.560**
Over all	0	0	0	0	0	400	400	8000	**10000**
	-	-	-	-	-	1153	1154	261.8	**1.278**

Table 6: **Experimental results on the 5.5HRS SAT2017 benchmark**.

Instance Class	CSoreSAT	Score$_2$SAT	YalSAT	ProbSAT	Dimetheus	Sparrow	gluHack	SparrowToRiss	BRSAP
	#suc	#suc	#suc	#suc	#suc	#suc	#suc	#suc	#suc
	par2	par2	par2	par2	par2	par2	par2	par2	par2
n=400	10	10	10	10	10	10	50	50	50
	8000	8000	8000	8000	8000	8000	9.507	164.8	**0.332**
n=420	20	20	20	20	20	20	50	50	50
	6000	6000	6000	6000	6000	6000	4.088	109.1	**0.932**
n=440	0	0	0	0	0	0	50	50	50
	-	-	-	-	-	-	7.289	205.5	**1.118**
n=460	10	10	10	10	10	10	50	50	50
	8000	8000	8000	8000	8000	8000	37.34	166.3	**1.184**
n=480	10	10	10	10	10	10	50	50	50
	8000	8000	8000	8000	8000	8000	33.63	158.0	**0.908**
n=500	20	20	20	20	20	20	50	50	50
	6001	6000	6000	6000	6000	6000	51.75	130.4	**1.230**
n=520	10	10	10	10	10	10	50	50	50
	8002	8000	8000	8000	8000	8000	39.30	173.1	**1.274**
n=540	0	10	10	10	10	10	50	50	50
	-	8000	8000	8000	8000	8000	42.69	190.9	**1.396**
Over all	80	90	90	90	90	90	400	400	400
	8000	7750	7750	7750	7750	7750	28.20	162.3	**1.047**

encouragingly, Table 4 shows that BRSAP is over 9 times faster than SparrowToRiss in overall 5.206HRS instances, indicating that BRSAP is the comprehensive best algorithm in this comparison. On the other hand, SparrowToRiss is the first place on the random SAT track of SAT Competition 2018 and gluHack also exhibits good performance on this benchmark, thus it is challenging to improve such performance over SparrowToRiss, indicating that BRSAP algorithm achieves the state-of-the-art performance on HRS instances with *r*=5.2.

7.1.4.4 Results on the 5.206HRS Random benchmark

To evaluate the performance of these solvers on large random HRS instances, we conduct the experiment of BRSAP and its state-of-the-art SLS competitors CSoreSAT, Score2SAT, YalSAT, ProbSAT, Dimetheus as well as Sparrow, and complete competitors gluHack as well as SparrowToRiss on the large random HRS ones with *r*=5.206. The experimental results are illustrated in Table 5. It is encouraging to see the performance of BRSAP remains surprisingly good on these 5.206HRS random benchmark, where its competitors show rather poor performance, especially for SLS solvers. It is apparent that BRSAP stands out as the best algorithm on this benchmark. According to Table 5, BRSAP consistently solves all HRS instances with up to 1000 instance, although the competitor SparrowToRiss solves 1600, 1200, 1200 runs on the n800, n900 and n1000 class respectively, whereas other all competitors fail to find a solution for any of these instances (CSoreSAT, Score$_2$SAT, YalSAT, Dimetheus, ProbSAT, Sparrow and gluHack), indicating the scalability of the BRSAP algorithm. Indeed, to the best of our knowledge, all 5.206HRS random benchmark are solved for the first time. Given the good performance of BRSAP on the 5.206HRS Random with 1000 variable, it is very likely it could be able to solve larger HRS instances with *r*=5.206.

7.1.4.5 Results on the 5.5HRS SAT2017 benchmark

Table 6 shows experimental results on the HRS instances with *r*=5.5. As is clear from Table 6, BRSAP shows significantly better performance than other competitors on the whole instances in terms of both successful runs and par 2. For the whole benchmark, BRSAP and SparrowToRiss succeed in all runs, while gluHack succeeds in 290 runs (out of 400 runs), and CSoreSAT succeeds in 80 runs, and Score$_2$SAT, YalSAT, ProbSAT, Dimetheus, and Sparrow 90 runs respectively. Particularly, the par 2 of BRSAP is about 155 times less than of SparrowToRiss, and about 7402 orders of magnitudes less than those of other state-of-the-art SLS competitors, indicating the effectiveness of BRSAP algorithm.

7.1.4.6 Results on the 5.5HRS Random benchmark

The experimental results for solving the large HRS instance with *r*=5.5 are presented in Table 7. It is clear that BRSAP shows significantly better performance than all its competitors on the whole benchmark. BRSAP is the only solver that solves these HRS instances with up to 1000 variables consistently (i.e., with 100% success rate), whereas all its competitors fail to find a solution for any of these instances with *n*=1000, and BRSAP outperforms its competitors in terms of par 2, which indicates the scalability of the BRSAP algorithm.

7.1.4.7 Results on the 5.699HRS Random benchmark

We conduct more empirical evaluations of BRSAP and its state-of-the-art SLS competitors CSoreSAT, Score$_2$SAT, YalSAT, ProbSAT, Dimetheus as well as Sparrow, and complete competitors gluHack as well as SparrowToRiss on HRS instances with *r*=5.699. The benchmark is generated by HRS tool [36].

The experimental results on the 5.699HRS benchmark are presented in Table 8. For n200 class, BRSAP is worse than gluHack, but BRSAP and gluHack solve the same number of instances. For n300, n400, n500, and n700 class, SparrowToRiss, gluHack and BRSAP show the same performance in terms of successful run, but BRSAP has less accumulative run time. For n800, n900 and n1000 instances, BRSAP stands out as the best solver in this comparison. Especially, BRSAP shows significantly superior performance than its competitors on n900 and n1000 class, where it solves all instances, while other competitors fail to find a solution for any of these instances. Overall, BRSAP solves 9000 instances, compared to 0, 0, 0, 0, 0, 0, 6000 and 2600 instances for CSoreSAT, Score$_2$SAT, YalSAT, ProbSAT, Dimetheus, Sparrow, gluHack and SparrowToRiss respectively, which clearly demonstrates the superiority of BRSAP over its SLS and complete competitors on solving HRS instances with *r*=5.699.

Table 7: **Experimental results on the 5.5HRS Random benchmark**.

Instance Class	CSoreSAT #suc par2	Score$_2$SAT #suc par2	YalSAT #suc par2	ProbSAT #suc par2	Dimetheus #suc par2	Sparrow #suc par2	gluHack #suc par2	SparrowToRiss #suc par2	BRSAP #suc par2
n=600	0	400	400	400	400	400	1600	2000	2000
	-	960.0	960.0	960.0	960.0	960.0	299.2	307.6	**1.376**
n=700	400	400	400	400	400	400	1200	800	**2000**
	963.3	960.0	960.0	960.0	960.0	960.0	596.2	760.8	**1.732**
n=800	0	400	400	400	400	400	0	400	**2000**
	-	960.0	960.0	960.0	960.0	960.0	-	960.3	**1.854**
n=900	400	400	400	0	400	400	0	800	**2000**
	986.2	960.0	960.0	-	960.0	960.0	-	816.8	**2.062**
n=1000	0	0	0	0	0	0	0	0	**2000**
	-	-	-	-	-	-	-	-	**2.202**
Over all	800	1600	1600	1200	1600	1600	2800	4000	**10000**
	1110	1008	1008	1056	1008	1008	899.1	758.7	**1.845**

Table 8: **Experimental results on the 5.699HRS Random benchmark**.

Instance Class	CSoreSAT #suc par2	Score$_2$SAT #suc par2	YalSAT #suc par2	ProbSAT #suc par2	Dimetheus #suc par2	Sparrow #suc par2	gluHack #suc par2	SparrowToRiss #suc par2	BRSAP #suc par2
n=200	0	0	0	0	0	0	1000	1000	1000
	-	-	-	-	-	-	**0.028**	46.19	0.224
n=300	0	0	0	0	0	0	1000	1000	1000
	-	-	-	-	-	-	0.459	101.4	**0.256**
n=400	0	0	0	0	0	0	1000	1000	1000
	-	-	-	-	-	-	2.589	229.2	**0.338**
n=500	0	0	0	0	0	0	1000	1000	1000
	-	-	-	-	-	-	42.08	249.8	**0.402**
n=600	0	0	0	0	0	0	800	800	1000
	-	-	-	-	-	-	275.7	470.6	**0.442**
n=700	0	0	0	0	0	0	1000	400	**1000**
	-	-	-	-	-	-	244.9	875.3	**0.786**
n=800	0	0	0	0	0	0	200	0	**1000**
	-	-	-	-	-	-	1070	-	**0.876**
n=900	0	0	0	0	0	0	0	0	**1000**
	-	-	-	-	-	-	-	-	**0.976**
n=1000	0	0	0	0	0	0	0	0	**1000**
	-	-	-	-	-	-	-	-	**1.056**
Over all	0	0	0	0	0	0	6000	2600	**9000**
	-	-	-	-	-	-	448.4	619.2	**0.595**

Table 9: The instances numbers, ratio and sizes for each HRS and URS with long clauses in the SAT2017 benchmark

	HRS			URS			
				5-SAT		7-SAT	
				medium	huge	medium	huge
#inst.	40	40	40	40	20	40	20
ratio	4.3	5.206	5.5	21.117	$r \in \{16.0, 16.2, \ldots, 19.8\}$	87.79	$r \in \{55.0, 56.0, \ldots, 74.0\}$
size	$n \in \{400, 420, \ldots, 540\}$			$n \in \{200, 210, \ldots, 590\}$	250000	$n \in \{90, 92, \ldots, 168\}$	50000

Table 10: The instances numbers, ratio and sizes for each HRS and URS with long clauses in the SAT2018 benchmark

	HRS			URS			
				5-SAT		7-SAT	
				medium	huge	medium	huge
#inst.	55	55	55	10	20	10	20
ratio	4.3	5.206	5.5	21.117	$r \in \{16.0, 16.2, \ldots, 19.8\}$	87.79	$r \in \{55.0, 56.0, \ldots, 74.0\}$
size	$n \in \{200, 220, \ldots, 400\}$			250	250000	120	50000

7.2 Experimental evaluation on URS and HRS

In order to show the generality and applicability of the proposed BRSAP algorithm, additional experiments on the URS and HRS benchmarks are carried out and the results are summarized in the following parts. Most (nearly 66.7% of) uniform instances in the benchmark of the random SAT track in SAT Competition 2017 are the ones at the phase transition. However, the performance of existing SLS algorithms on random k-SAT instances at the phase transition is still unsatisfactory. Thus, results of extensive experiments to evaluate BRSAP on uniform k-SAT instances at the phase transition and with long clauses are provided.

7.2.1 Benchmarks and Experiment Preliminaries

All the URS instances used in our experiments are generated according to the and k-SAT generator[6] . We adopt the following 4 testing benchmarks.

1) **SAT2017**: all 120 HRS instances and all 120 medium and huge random k-SAT instances with long clauses from SAT Competition 2017, and each k-SAT, the instances contains various sizes and ratios. The details of the benchmark are given in Table 9.
2) **URS 5-SAT**: Random 5-SAT problems generated by the k-SAT generator. Medium 5-SAT instances at the threshold ratio of phase transition (r=21.115, 100 instances, n=200, 250, 300, 350, 400, 20 instances for each size)
3) **URS 7-SAT**: Random 7-SAT problems generated by the k-SAT generator. Medium 7-SAT instances at the threshold ratio of phase transition (r=87.79, 100 instances, n=110, 120,130, 140, 150, 20 instances for each size)
4) **SAT2018**: all 165 HRS instances and all 60 medium and huge random k-SAT instances with long clauses from SAT Competition 2018[7]. The details of the benchmark are given in Table 10.

We tuned the β, η and γ parameters of BRSAP on URS according to our experience in Table 11.

[6]https://sourceforge.net/projects /ksat generator/
[7]http://sat2018.forsyte.tuwien.ac.at/

Table 11: Parameter settings of BRSAP for URS instances

scale	5-SAT 7-SAT
medium instances	β=100000
	μ=1000
	η=1000
huge instances	β=5000
	μ=50
	η=50

The complete solvers did not solve any instances for the medium and huge instances of the SAT competition in 2018 (except the champion solver SparrowToRiss), thus, gluHack was not applied to solve the medium and huge random k-SAT instances in the following experiments. In order to evaluate the relative effectiveness and efficiency of BRSAP, we compare BRSAP with SparrowToRiss, CScoreSAT, Score$_2$SAT, YalSAT and PobSAT on URS and HRS benchmarks.

Experiments on the four benchmarks are carried out on Intel(R) Core (TM) i7-6700U 3.4 GHz CPU with 16GB RAM, running the 64-bit Ubuntu Linux operating system. The CPU time limit is 5000 seconds. For all benchmarks, each solver is executed 10 times for each instance. we report average solved instances at ten run “AverS” for these benchmarks as well as “par 2”. The best results for an instance class are highlighted in **bold**. If a solver has no successful run on an instance class, the corresponding “par 2” is marked with “-”.

7.2.2 Experimental Results

In the following, we present the comparative experimental results of BRSAP and its competitors on each benchmark.

7.2.2.1 Results on the SAT2017 benchmark

Table 12 presents the results of the performance of BRSAP compared with state of the art SLS solvers on all HRS and URS with long clauses from SAT Competition 2017. The results show that for 5-SAT instances with r=21.117, the performance of BRSAP, Score$_2$SAT and CScoreSAT are similar and better than that of other competitors, and for the remaining instances class, BRSAP significantly outperforms its competitors in terms of metrics.

Especially, BRSAP succeeds in a few more average runs than its competitors on random 7-SAT instances at phase

Table 12: **Experimental results on the SAT2017 benchmark**.

Random SAT	r	SparrowToRiss		CScoreSAT		Score$_2$SAT		YalSAT		PobSAT		BRSAP	
		AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2
HRS	4.3	40	0.117	40	0.009	40	**0.008**	40	0.017	40	0.057	40	0.115
	5.206	40	5.709	0	-	0	-	0	-	0	-	40	**0.594**
	5.5	40	151.0	6	8500	9	7750	9	7750	9	7750	40	**0.980**
URS	<21.117	4	8083	10	5250	8	6231	12	4147	11	4526	**13**	**3805**
	21.117	9	7760	**15**	**6476**	14	6655	13	6880	13	6829	14	6667
	<87.79	9	5602	11	4839	11	5756	9	5517	11	4514	**12**	**4082**
	87.79	16	6035	18	5931	19	5582	17	5957	18	5552	**21**	**4993**
Overall/240		158	3466	100	5992	101	5997	100	5903	102	5775	**180**	**2801**

Table 13: **Experimental results on the URS 5-SAT benchmark**.

Ratio	Variable	SparrowToRiss		CScoreSAT		Score$_2$SAT		YalSAT		PobSAT		BRSAP	
		AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2
r=21.117	n=200	11	4516	11	4506	11	4523	11	4513	11	4513	11	**4502**
	n=250	9	5582	10	5069	9	5502	10	5247	10	5142	10	**5112**
	n=300	3	8525	8	6298	9	6078	**10**	**5283**	8	6122	9	5894
	n=350	8	6091	12	4166	13	3749	13	3734	13	3734	13	**3721**
	n=400	1	9510	3	8667	3	8728	2	9216	3	8613	3	**8602**

Table 14: **Experimental results on the URS 7-SAT benchmark**.

Ratio	Variable	SparrowToRiss		CScoreSAT		Score$_2$SAT		YalSAT		PobSAT		BRSAP	
		AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2
r=87.79	n=110	10	5087	10	5141	11	4592	11	4749	11	4559	11	**4532**
	n=120	9	5626	9	5780	10	5248	10	5451	9	5969	10	**5261**
	n=130	10	5123	10	5518	**13**	**3981**	13	4412	12	4380	11	4774
	n=140	10	5087	11	4829	13	4048	10	5397	10	5597	13	**4019**
	n=150	0	-	3	8594	4	8327	5	8119	5	7965	**7**	**7394**

Table 15: **Experimental results on the SAT2018 benchmark**.

Random SAT	Ratio	SparrowToRiss		CScoreSAT		Score$_2$SAT		YalSAT		PobSAT		BRSAP	
		AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2	AverS	par 2
HRS	4.3	55	0.052	55	0.009	55	**0.001**	55	**0.001**	55	0.013	55	0.012
	5.206	55	1.020	8	8591	33	4000	9	8387	12	7858	55	**0.324**
	5.5	55	136.4	11	8000	12	7818	12	7818	12	7818	55	**0.516**
URS	<21.117	3	8570	9	5706	11	4683	12	4079	11	4524	**13**	**3941**
	21.117	7	3111	8	2495	7	3015	8	2326	7	3404	**9**	**1523**
	<87.79	9	5657	10	5129	11	4720	9	5520	11	4522	**12**	**4118**
	87.79	8	**2262**	5	5224	8	2453	6	4488	8	2967	8	2692
Overall/225		192	1537	106	5362	137	3968	111	5117	116	4919	**207**	**784.3**

transition. BRSAP succeeds in 21 average runs, compared to 19 for Score$_2$SAT, and 18 for both ProbSAT and CScoreSAT, and 17 for YalSAT, and 16 for SparrowToRiss. Further observation shows that BRSAP succeeds in 179 average runs, compared to 158 for SparrowToRiss, and 102 for ProbSAT, and 101 for Score$_2$SAT, and 100 for both CScoreSAT and YalSAT. Overall, BRSAP succeeds in 180 average runs, whereas none of its competitors succeeds in more than 160 average runs with the half cutoff time, which illustrates its robustness and scalability.

7.2.2.2 Results on the URS 5-SAT benchmark

To measure the performance of BRSAP on URS instances at phase-transition more accurately, we additionally test BRSAP on the medium 5-SAT instances. The results are presented in Table 13. According to the Table 13, BRSAP has similar performance with ProbSAT, CScoreSAT, YalSAT and Score$_2$SAT on this benchmark.

7.2.2.3 Results on the URS 7-SAT benchmark

In order to measure the performance of BRSAP on 7-SAT instance at phase transition, we compare BRSAP with ProbSAT, CScoreSAT, YalSAT, SparrowToRiss which is the best SLS solver in the random track of SAT Competitions in 2018, and Score$_2$SAT which is the best SLS solver on URS at phase-transition in the random track of SAT Competitions in 2017. The results are reported in Table 14. As can be seen from Table 14, BRSAP does not give the best performance on the only 7-SAT instance with n=130, but BRSAP has similar performance to the solvers SparrowToRiss and Score$_2$SAT.

7.2.2.4 Results on the SAT2018 benchmark

To investigate the performance of BRSAP on URS and HRS benchmarks with various ratio, we compare it with ProbSAT, CScoreSAT, YalSAT, SparrowToRiss and Score$_2$SAT on all HRS instances and URS instances with long clauses from SAT

Competition 2018. Table 15 summarizes the experimental results on the SAT2018 benchmark.

BRSAP gives the best performance for all random SAT instances except for the HRS instances with *r*=4.3 and URS instances with *r*=87.79, and especially it solves more HRS instances than all SLS competitors and more URS instances with long clauses than all competitors. For the URS instances with *r*=87.79, BRSAP solves as many instances as SparrowToRiss, but the par 2 is a little more than SparrowToRiss's. Overall, BRSAP solves 207 instances on average, and SparrowToRiss solves 192 instances on average, and Score$_2$SAT solves 137 instances on average, and ProbSAT solves 116 instances on average, and YalSAT solves 111 instances on average, and CScoreSAT solves 106 instances on average. BRSAP significantly outperforms SparrowToRiss on all random SAT instances. SparrowToRiss is the first place on the random SAT track of SAT Competition 2018, thus it is challenging to improve such performance over SparrowToRiss, indicating that BRSAP algorithm achieves the state-of-the-art performance on random SAT instances, which illustrates the robustness and scalability of BRSAP algorithm on HRS instances and URS instances with long clauses.

7.3 Summary of experimental results

According to these experiments including in Tables 2-8, BRSAP is significantly better than the state-of-the-art SLS solvers and complete solvers on a broad range of instances, and shows the efficiency and the robustness on solving all testing HRS instances with up to 1000 variables. This experiment clearly demonstrates that the superiority of BRSAP becomes more significant over its competitors as the size of HRS instances increases. As can be seen from Tables 12-15, BRSAP is quite competitive for solving URS with long clauses. Thus, BRSAP can effectively solve both URS problems with long clauses and HRS problems (The current state-of-the-art SLS solvers can only effectively solve URS instances, and complete solvers can only effectively solve HRS instances. There is no solver that can effectively solve both HRS and URS).

Moreover, the heuristics used by SLS solvers to solve random SAT problems are also potentially useful for solving real-world SAT problems. The SAT instances encoded from real-world applications may be of large size. Therefore, it is of great significance to develop a fast and efficient SAT solver solving theories and methods. Also, SLS is an efficient method for solving graphs, gene regulatory networks, automated verification, scheduling and computing theory. In this work, our BRSAP algorithm is able to solve large HRS instances with up to 1000 variables within five seconds and can effectively solve URS with long clauses, and thus can provide support for solving problems from the application domain.

8 Discussions

Some further discussions are given below to clarify some issues and highlight some important cases.

8.1 Effectiveness of the BRSAP components

In this section, we present a detailed discussion on each underlying component of BRSAP algorithm, namely GWU, GWAC, second-level-biased random walk strategy, the new tie-breaking strategy, the *score* property and the *age* greed property. Since almost all state-of-the-art SLS solvers can effectively solve all HRS instances with *r*=4.3, we do extensive experiments for following alternative versions on all testing HRS benchmarks expect for the HRS instances with *r*=4.3. The computing environments for these experiments are the same as those used for experiments in Section 7.1.

8.1.1 Effectiveness of GWU

In order to demonstrate the effectiveness of clause weighting scheme GWU in the BRSAP algorithm, we conduct experiments to compare BRSAP with its an alternative version named BRSAP_alt1, which does not utilize the GWU, i.e., removing update clause' weights GWU of lines 9-11 and removing update GWU of line 23 in Algorithm 1. We use the default value of BRSAP as the parameter settings of η and γ. The BRSAP_alt1 algorithm solves six testing benchmarks and performs ten times for each instance with the cutoff time of 600 seconds. The experimental results on the six benchmarks are shown in Table 16.

From the results in Table16, it is apparent that BRSAP_alt1 fails to solve any instance with *r*=5.206 and *r*=5.699, and BRSAP_alt1 succeeds in solving 50 runs (out of 400), 400 runs (out of 10000) and 760 runs (out of 1650). The performance of BRSAP significantly outperforms that of BRSAP_alt1, demonstrating the significance of the clause weighting scheme GWU (i.e., the significance of HSCs-GWU)

8.1.2 Effectiveness of GWAC

The BRSAP algorithm does not use the new clause weighting scheme GWAC, i.e., removing lines 12-14 and 16 in Algorithm 1. We obtain an alternative degenerating version called BRSAP_alt2. We use the default value of BRSAP as the parameter settings of β and γ.

We conduct extensive experiments to show the effectiveness of GWAC on all testing instances. The BRSAP_alt2 also performs ten times for each instance with the cutoff time of 600 seconds. The experimental results on the six benchmarks are reported in Table 16.

In terms of success runs, BRSAP significantly outperforms BRSAP_alt2 on all six benchmarks. The instance class for which BRSAP does not give the best performance is HRS instances with *r*=5.206 and *r*=5.5 in terms of par2. Although BRSAP_alt2 spends less time than BRSAP, BRSAP and BRSAP_alt2 solve the same instances with *r*=5.206 and *r*=5.699. For the HRS instances with *r*=5.5, we observe that BRSAP significantly outperforms BRSAP_alt2. The improvement of BRSAP over BRSAP_alt2 is small, but the gap is still considerable on the HRS instances with *r*=5.5. Overall, the comparison between BRSAP and BRSAP_alt2 shows that updating the clause' weights in BRSAP is of great significance for solving HRS instances with *r*=5.5.

Table 16: Comparison among BRSAP and its alternative degenerating versions on the six testing benchmarks. Each solver is performed 10 times on each class, and the results in bold indicate the best performance for each class.

Benchmarks	BRSAP	BRSAP_alt1	BRSAP_alt2	BRSAP_alt3	BRSAP_alt4	BRSAP_alt5	BRSAP_alt6	BRSAP_alt7
	suc	suc	suc	suc	suc	suc	suc	suc
	par2	par2	par2	par2	par2	par2	par2	par2
5.206HRS	400	0	400	0	400	390	400	0
SAT2017	0.671	-	**0.545**	-	0.884	30.64	0.675	-
5.206HRS	10000	0	10000	0	10000	8400	10000	0
Random	1.278	-	**0.989**	-	1.498	193.2	1.287	-
5.5HRS	**400**	50	380	90	130	370	370	90
SAT2017	**1.047**	1050	60.72	930.0	810.2	90.92	90.93	930.0
5.5HRS	**10000**	400	9200	800	1200	8400	9200	1200
Random	**1.845**	1152	97.38	1104	1056	193.5	97.69	1056
HRS	**1650**	760	1600	770	1300	1580	1590	760
SAT2018	**0.173**	652.5	36.50	647.2	254.7	51.06	43.80	648.6
5.699HRS	9000	0	9000	0	8200	8600	8400	0
Random	0.595	-	**0.480**	-	107.3	53.89	80.57	-

8.1.3 Effectiveness of the second-level-biased random walk

By removing all clause weighting schemes, i.e., removing the GWU (i.e., removing update clause' weights GWU of line 23 in Algorithm 1) and GWAC (i.e., removing update clause' weights GWU of line 16 in Algorithm 1), i.e., replacing the biased random walk component, i.e., lines 9-15 in Algorithm 1, with the standard random walk component, i.e., line 15 in Algorithm 1), we obtain this alternative version named BRSAP_alt3. BRSAP_alt3 utilizes the default value of BRSAP as the parameter settings of γ.

We conduct a large number of experiments to show the effectiveness of biased random walk on the six benchmarks, and the results are summarized in Table 16. The BRSAP_alt3 performs ten times for each instance with the cutoff time of 600 seconds.

The experimental results show that BRSAP obviously outperforms BRSAP_alt3. Specifically, BRSAP_alt3 fails to solve any instance with r=5.206 and r=5.699, which indicates that the importance of the biased random walk based on GWU and GWAC.

8.1.4 Effectiveness of the new tie-breaking strategy

In this subsection, we do more experiments to analyze the effectiveness of the new tie-breaking strategy (lines 18-20 in Algorithm 1) in the BRSAP algorithm. To demonstrate the effectiveness of the new tie-breaking strategy, we do not utilize the tie-breaking strategy, i.e., removing lines 18-20 in Algorithm 1. We obtain an alternative degenerating version called BRSAP_alt4, which allows the same variable to be selected in successive steps. We use the default value of BRSAP as the parameter settings of η and β.

We evaluate BRSAP_alt4 on six testing benchmarks and the results are shown in Table 16, where each solver performs 10 times with a cutoff time of 600 seconds.

BRSAP shows significantly better performance than BRSAP_alt4 on the all six benchmarks in terms of both successful runs and average run time. Particularly, on the 5.5HRS SAT2017, 5.5HRS Random, HRS SAT2018 and 5.699HRS Random benchmarks, the runtime of BRSAP is about 774 times, 572 times, 1472 times and 180 times less than of BRSAP_alt5 respectively. The results confirm the effectiveness of the new tie-breaking as does in BRSAP on solving HRS instances.

8.1.5 Effectiveness of the greedy property score

This alternative version of BRSAP utilizes the tie-breaking strategy, but the *SA* function only uses *age* (i.e., replacing the *SA* function, i.e., *SA* of line 19 in Algorithm 1, with the *age*). Thus, we obtain this alternative version called BRSAP_alt5, which uses the default value of BRSAP as the parameter settings. BRSAP_alt5 is executed ten times on each instance with the cutoff time of 600 seconds.

From the results of Table 16, it is clear that BRSAP significantly outperforms BRSAP_ alt4 on all HRS instances, which indicates that if we do not utilize the greedy property *score* as does in BRSAP, the algorithm performs much worse than BRSAP.

8.1.6 Effectiveness of the diversification property age

By removing the *age* in the BRSAP algorithm, i.e., replacing *SA* with only *score* in line 19 in Algorithm 1, we obtain an alternative degenerating version named BRSAP_alt6, which uses the default value of BRSAP as the parameter settings of η and β. BRSAP_alt6 is executed ten times on each instance with the cutoff time of 600 seconds.

According to the results of Table 16, the performance of BRSAP significantly outperforms that of BRSAP_ alt6 on all six HRS benchmarks. Specially, BRSAP_ alt6 succeeds in solving 8400 runs on 5.699HRS Random benchmark, whereas BRSAP_ alt5 and BRSAP succeed in solving 8600 runs and 9000 runs on 5.699HRS Random benchmark respectively, which indicates that the importance of property *age*.

8.1.7 Effectiveness of clause weighting schemes and tie-breaking strategy

This alternative version of BRSAP does not use GWU, GWAC and the new tie-breaking strategy. i.e., does not utilize biased random walk strategy and the *SA* (i.e., removing lines 9-14, 16, 18-20 and 23 in Algorithm 1, i.e., only using the polynomial probability and standard random walk). This alternative version is named BRSAP_alt7, which is no parameter to be set.

We evaluate BRSAP_alt7 on all six benchmarks, where each solver performs ten tines with a cutoff time of 600 seconds.

Table 16 presents that BRSAP_alt7 fails to solve any instances with r=5.206 and r=5.699; even on the 5.699HRS Random benchmark including instances with n=200. The performance of BRSAP is obviously better than that of BRSAP_alt7, conforming the significance of the new clause weighting schemes and the new tie-breaking strategy.

8.2 Approximate Implementation of BRSAP

In this paper, the implementation of BRSAP described in Sections 4-6.

Inspired by the approximate implementation of the SWT strategy [39], we firstly propose an accurate implementation of GWU scheme, which updates the weights of unsatisfied clauses during the search process. The maintenance of the accurate implementation is described as follows: whenever a variable x is flipped during the search, each clause $c \in C(x)$ ($C(x)=\{c \mid c$ is a clause which x appears in $\boldsymbol{c}\}$) is checked whether c's state is changed (from unsatisfied to satisfied, from satisfied to unsatisfied) by flipping a variable x (the implementation of checking clauses' state on BRSAP is equal to one on probability SLS algorithms like ProbSAT [17]). If it is the case (c's state is unsatisfied by flipping the variable y), c's GWU value is updated.

Note that the discussions below are based on the condition that F is a random k-SAT instance with n variables and m clauses ($r=m/n$). For each clause c, the number of all variables is equal to k, i.e., $E(|c|) = k$. We use $F(s)$ to denote the number of unsatisfied clauses in step s, thus $E(|F(s)|) < m$.

For the accurate implementation of GWU scheme, the time complexity of computing the unsatisfied clauses' GWU at step s is $O(E(|F(s)|)) < O(m)$.

Inspired by the approximate implementation of the *age* function [30], we propose an accurate implementation of GWAC scheme, which updates the weights of clause selected during the search process, i.e., only one clause's GWAC value is updated at each step, thus for the accurate implementation of GWAC scheme, the time complexity of computing the selected clause' GWAC at each step is $O(1)$.

The second-level-biased random walk strategy is based on the idea of biased random walk strategy [45]. However, the second-level-biased random walk strategy is utilized to select a clause from two sets (HSCs-GWU and LDCs-GWAC) in the worst case. HSCs-GWU and LDCs-GWAC are updated by the unsatisfied clauses at each step. For the accurate implementation of second-level-biased random walk strategy described in Section 5, the worst-case time complexity of selecting an unsatisfied clause at step s is $O(E(|F(s)|)) + O(E(|F(s)|)) = O(E(|F(s)|))$.

The probability strategy is utilized to select a variable from the unsatisfied clause c selected based on the second-level-biased random walk strategy. The approximate implementation of probability strategy on BRSAP is equal to one on SLS algorithms based on probability strategy like ProbSAT. The new tie-breaking strategy based on the new function *SA* is that the last flipping variable must not be the current flipping variable. The tie-breaking strategy is also used to select a variable from the unsatisfied clause c selected based on the second-level-biased random walk strategy. Thus, for the accurate implementation of variable selection heuristic, the worst-case time complexity of computing the probability strategy and tie-breaking strategy is $O(E(|c|)) + O(E(|c|)) = O(E(|c|)) = O(k)$.

Compared with SLS algorithms only based probability strategy like ProbSAT, the additional implementations of BRSAP are the second-level-biased random walk strategy and the new tie-breaking strategy. Thus, the worst-case time complexity of adding the implementations is $O(E(|F(s)|)) + O(E(|c|)) = O(E(|F(s)|)) + O(k) < O(m) + O(k)$.

According to the literature [62], it shows that all the time complexities of SLS algorithms only based probability strategy (like PrboSAT) are about $O(k*r)$. Thus, all the time complexities of the approximate implementation of BRSAP are about $O(E(|F(s)|)) + O(E(|c|)) + O(k*r) = O(E(|F(s)|)) + O(k) + O(k*r) = O(E(|F(s)|)) + O(k*r)$. If the number of unsatisfied clauses is not greater than $k*r$ in step s, then the time complexities of the approximate implementation of BRSAP are about $O(k*r)$. Otherwise, the time complexities of the approximate implementation of BRSAP are greater than $O(k*r)$. According to our experience, when the algorithm executes after larger than a certain step s, the number of unsatisfied clauses must be less than or equal to $k*r$ (This conclusion needs to be proved later). Thus, the time complexities of the approximate implementation of probability strategy are close to those of the approximate implementation of BRSAP.

The existing probability strategy is ineffective when solving to HRS, while the second-level-biased random walk strategy and the new tie-breaking strategy shows effectiveness when applying to probability strategy, and the related empirical analyzes have be shown in Sections 8.1-8.8. The possible reason is that second-level-biased random walk strategy and the new tie-breaking strategy help probability algorithms to decrease blind unreasonable search and thus leads probability SLS algorithms to promising search spaces.

9 Conclusions and future work

In this work, we proposed two new global clause weighting schemes GWU and GWAC and a new scoring function *SA* based on greedy property *score* and diversification property *age* for improving SLS algorithms on SAT instances, resulting in an effective SLS algorithm namely BRSAP, which shows excellent performance on HRS instances and URS instances.

The main results are summarized below:

1) Firstly, only considering unsatisfied clauses, we proposed a global clause weighting scheme named GWU, which aims to distinguish unsatisfied clauses. We also defined hard satisfiable clauses and easy satisfiable clauses accordingly.

2) In order to distinguish easy satisfiable clauses, based on the current clauses selected, we further proposed another global clause weighting scheme called GWAC. Then we also defined *long age clauses* and *short age clauses* accordingly.

3) Based on the GWU and GWAC, we developed a second-level-biased random walk strategy to select a clause.

4) Finally, in order to prevent the same variable to be selected in consecutive steps, we adopted the tie-breaking strategy, but the previous tie-breaking strategy is not suitable for HRS instances. Thus, we proposed the *SA* function combining the *score* (greedy property) and *age* (diversification property), which is utilized to break ties. Finally, second-level-biased random walk strategy based on two global clause weighting schemes and a new scoring function were used to develop the BRSAP algorithm.

BRSAP's effectiveness has been demonstrated on random SAT problems from the SAT Competitions in 2017 and 2018, and on randomly generated HRS and URS with long clauses problems. The results show that BRSAP outperforms state-of-the-art SLS solvers and the state-of-the-art complete solver in most cases. Moreover, BRSAP can effectively solve both URS problems and HRS problems.

Further investigations show that the effectiveness of BRSAP is attributed to second-level-biased random walk strategy based on two global clause weighting schemes and the tie-breaking strategy based on a linear scoring function *SA*, especially the clause weighting scheme GWU.

The heuristics used by SLS solvers to solve random SAT problems are also potentially useful for solving real-world SAT problems [47-49]. The SAT instances encoded from real-world applications may be of large size. As our BRSAP algorithm is able to solve large HRS instances quickly with up to 1000 variables within five seconds, and may be beneficial to solving cryptography instances, and thus we believe the experimental results of BRSAP on HRS instances and URS instances may provide support for solving problems from the application domain.

For future work, we plan to combine the global clause weighting schemes and the new tie-breaking strategy with other algorithmic techniques, such as linear make [25] and configuration checking [1], [3]. Also, inspired by the success of two global clause weighting schemes based on GWU and GWAC, we would like to explore more global clause weighting schemes, and thus employ them to develop more efficient SLS algorithms for random SAT. Additionally, we would like to apply the GWU, GWAS, the scoring function *SA* to improving performance of SLS algorithms on solving the structured instances in SAT competition.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (Grant No.61673320) and the Fundamental Research Funds for the Central Universities (Grant No.2682019ZT16 and No.2682020CX59), and Sichuan Science and Technology Program (Grant No. 2020YJ0270). The authors would like to thank Tomáš Balyo for providing the HRS generator.

REFERENCES

[1] C. Luo, K. Su and S. Cai (2014). More efficient two-mode stochastic local search for random 3-satisfiability[J]. Applied Intelligence, vol. 41, no. 3, pp.665-680.

[2] H. Fu, Y. Xu, G. Wu, H. Jia, W. Zhang and R. Hu, "An Improved Adaptive Genetic Algorithm for Solving 3-SAT Problems Based on Effective Restart and Greedy Strategy," *Inter. J.Com. Intell. Sys.*, vol. 11, no. 1, pp.402-413, Jan. 2018.

[3] M. Davis, and H. Putnam (1960). "A computing procedure for quantification theory," J. ACM, vol.7, no. 3, pp: 201-215.

[4] M. Davis, G. Logemann, and D. W. Loveland (1962), "A machine program for theorem-proving," Commun. ACM, vol. 5, no. 7, pp. 394–397.

[5] C. Weidenbach, D. Dimov, A. Fietzke, et al (2009). "Wischnewski P. SPASS Version 3.5," In: Proceedings of Automated Deduction, Springer, Berlin, Heidelberg, pp. 140-145.

[6] S. Cai and K. Su (2011)."Local search with configuration checking for SAT," In: Proceedings of ICTAI, pp. 59–66.

[7] C. Luo, S. Cai, K. Su and W. Wu (2015). "Clause states based configuration checking in local search for satisfiability," *IEEE Trans. Cybern*, vol. 45, no. 5, pp. 1028-1041.

[8] H. H. Hoos and T. Stützle (2004), Stochastic Local Search: Foundations & Applications. San Francisco, CA, USA: Elsevier/Morgan Kaufmann, Sep. 2004.

[9] M. Mavrovouniotis, F. M. Müller and S. Yang (2017), "Ant colony optimization with local search for dynamic traveling salesman problems," IEEE Trans. Cybern, vol. 47, no. 7, pp. 1743-1756.

[10] T. Balyo (2016). Using algorithm configuration tools to generate hard random satisfiable benchmarks. In: Proceedings of SAT 2016, pp. 60–62.

[11] B Selman, H A Kautz and B Cohen (1994). "Noise strategies for improving local search". In: Proceedings of AAAI, pp. 337–343.

[12] H H Hoos (2002). "An adaptive noise mechanism for WalkSAT". In: Proceedings of AAAI, pp. 655–660.

[13] O. Gableske (2018). Dimetheus. In: Proceedings of SAT 2018, pp. 20-21.

[14] Yin L, He F, Hung WNN, Song X, Gu M (2012) Maxterm covering for satisfiability. IEEE Trans Comput 61(3):420–426.

[15] S. Liu and A. Papakonstantinou. "Local search for hard sat formulas: the strength of the polynomial law," *in 30th AAAI Conf. Artif. Intell.*, Feb. 2016, pp. 732-738.

[16] O Gableske (2016). "Sat solving with message passing". PhD dissertation, Ulm University, Germany, 2016.

[17] A Balint and U Schöning (2018). "probSAT". In: Proceedings of SAT 2018, pp. 35.

[18] S Cai, K Su and C Luo (2013). "Improving walksat for random k-satisfiability problem with k>3". In: Proceedings of AAAI, pp. 145-151.

[19] C Luo, S Cai, W Wu and K Su (2014). "Double configuration checking in stochastic local search for satisfiability". In: Proceedings of AAAI, pp. 2703-2709.

[20] A. Biere (2017). "CADICAL, LINGELING, PLINGELING, TREENGELING and YALSAT: Solver description," In: Proceedings of SAT 2017, pp. 14-15.

[21] C. Luo, S. Cai, W. Wu, and K. Su (2016). CSCCSat2014. In: Proceedings of SAT 2016, pp, 10.

[22] S. Cai and C. Luo (2017). "Score$_2$SAT: Solver description," In: Proceedings of SAT 2017, pp. 34.

[23] A. Balint and N. Manthey (2018). SparrowToRiss. In: Proceedings of SAT 2018, pp, 38-39.

[24] A. Zha (2018). GluHack. In: Proceedings of 2018, pp, 26.

[25] S Cai, C Luo and K Su (2014). Improving walksat by effective tie-breaking and efficient implementation. Computer Journal, 58(11): 2864-2875.

[26] A. Balint and U. 7 (2012), "Choosing probability distributions for stochastic local search and the role of make versus break," In: Proceedings of SAT 2012, pp. 16–29.

[27] H. H. Hoos and T. Stützle (2000). "Local search algorithms for SAT: An empirical evaluation," J. Autom. Reasoning, 24(4), pp. 421–481.

[28] C. M. Li and W. Q. Huang (2005). "Diversification and determinism in local search for satisfiability," In: Proceedings of SAT 2005, pp. 158–172.

[29] D.A. Tompkins and H.H. Hoos (2010). "Dynamic scoring functions with variable expressions: New SLS methods for solving SAT," In:

Proceedings of SAT 2010, pp. 278-292.

[30] S. Cai, C. Luo and K. Su (2014). "Scoring functions based on second levell score for *k*-SAT with long clauses." Jour. Artif. Intell. Resea., vol. 51, no. 2014, pp. 413-441.

[31] A. Balint and A. Fröhlich (2010), “Improving stochastic local search for SAT with a new probability distribution,” in Proc. SAT, Edinburgh, U.K., Jul. 2010, pp. 10–15.

[32] S. Cai, Z. Jie and K. Su. (2015). "An effective variable selection heuristic in SLS for weighted Max-2-SAT." Journal of Heuristics, vol. 21, no. 3, pp. 433-456.

[33] S. Cai. (2015). "Balance between Complexity and Quality: Local Search for Minimum Vertex Cover in Massive Graphs." In: Proceedings of AAAI, pp. 747-753.

[34] H. Zhang, S. Cai, et al (2017). "An efficient local search algorithm for the winner determination problem." Journal of Heuristics, vol. 23, no. 2, pp. 1-30.

[35] C. Luo, H. Hoo, S. Cai, et al (2019). “Local Search with Efficient Automatic Configuration for Minimum Vertex Cover”. In: Proceedings of IJCAI, pp. 1297-1304.

[36] T. Balyo and L. Chrpa (2018). “Using Algorithm Configuration Tools to Generate Hard SAT Benchmarks”. In: Proceedings of SoCS 2018, pp,133-137.

[37] F. Hutter, H. H. Hoos, and K. Leyton-Brown (2011). “Sequential modelbased optimization for general algorithm configuration”. In: Proceedings of LION 2011, pp, 507–523.

[38] W. Barthel, A. K. Hartmann, M. Leone, F. Ricci-Tersenghi, M. Weigt, and R. Zecchina (2002). “Hiding solutions in random satisfiability problems: A statistical mechanics approach”. Physical review letters, vol. 88, no. 18, pp. 188701.

[39] S. Cai and K. Su (2013). "Local search for Boolean Satisfiability with configuration checking and subscore." Artificial Intelligence vol. 204, no. 2013, pp.75-98.

[40] Z. Wu and B. W. Wah (2000). “An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems”. In: Proceedings of AAAI, pp. 310-315.

[41] J. Thornton (2005). “Clause weighting local search for SAT”. Journal of Automated Reasoning, vol. 35, no. 1-3, pp. 97-142.

[42] F. Hutter, D. A. Tompkins and H. H. Hoos (2002). “Scaling and probabilistic smoothing: Efficient dynamic local search for SAT”. In: Proceedings of CP 2002, pp. 233-248.

[43] D. Liang, Y. Wu and S. Ma (1998). “An efficient local search algorithm for structured SAT problems”. Chinese Journal of computers, vol. s1, pp. 92-97.

[44] J. Qiu, Y. Zhang (2010). “A Heuristic Algorithm for Solving the Satisfiability Problem Based on the Key Literal”. Computer & Digital Engineering, 10.

[45] C. Luo, S. Cai, K. Su and W. Huang (2017). “CCEHC: An efficient local search algorithm for weighted partial maximum satisfiability”. Artificial Intelligence, vol. 243, pp. 26-44.

[46] H. Fu, S. Chen, Y. Xu and J. Liu (2020). “Improving WalkSAT for Random 3-SAT Problems”. Journal of Universal Computer Science, vol. 26, no. 2, pp. 220-243.

[47] C. Bright, K. Ilias, and G. Vijay (2019). "Applying computer algebra systems with SAT solvers to the Williamson conjecture." Journal of Symbolic Computation, vol. 100. no. 2020, pp. 187-209.

[48] B. König, N. Maxime, and N. Dennis (2018). "CoReS: A Tool for Computing Core Graphs via SAT/SMT Solvers." In: Proceedings of Graph Transformation, pp. 37-42.

[49] A. Deshpande, and R. K. Layek (2019). “Fault detection and therapeutic intervention in gene regulatory networks using SAT solvers”. BioSystems, vol. 179, pp.55-62.

[50] Z. Lei, and S. Cai (2019). "NuDist: An Efficient Local Search Algorithm for (Weighted) Partial MaxSAT." The Computer Journal. bxz063, https://doi.org/10.1093/comjnl/bxz063.

[51] Y. Fu, Z. Lei, S. Ca, J. Lin, and H. Wang, (2020). “WCA: A weighting local search for constrained combinatorial test optimization”. Information and Software Technology, 122, 106288.

[52] Z. Lei, and S. Cai (2018). “Solving (Weighted) Partial MaxSAT by Dynamic Local Search for SAT”. In: Proceedings of IJCAI, pp. 1346-1352.

[53] S. Cai, J. Lin, and K. Su (2015). “Two weighting local search for minimum vertex cover”. In: Proceedings of AAAI 2015, pp 1107–1113.

[54] M., Ouimet, and K. Lundqvist, (2007). “Automated verification of completeness and consistency of abstract state machine specifications using a sat solver”. Electronic Notes in Theoretical Computer Science, 190(2), 85-97.

[55] X. Zhao, L. Zhang, D. Ouyang and Y. Jiao, (2009). “Deriving all minimal consistency-based diagnosis sets using SAT solvers”. Progress in Natural Science, 19(4), 489-494.

[56] J. Coelho, & M. Vanhoucke, (2011). “Multi-mode resource-constrained project scheduling using RCPSP and SAT solvers”. European Journal of Operational Research, 213(1), 73-82.

[57] V. Ulyantsev and F. Tsarev, (2012). “Extended finite-state machine induction using SAT-solver”. IFAC Proceedings Volumes, 45(6), 236-241.

[58] J.P. Marques-Silva, I. Lynce, S. Malik (2009). “Conflict-Driven Clause Learning SAT Solvers”. Frontiers in Artificial Intelligence & Applications, 185(4), pp.131-153.

[59] M.J.H., Heule, and H., van Maaren (2009).” Look-Ahead Based SAT Solvers”. Handbook of Satisfiability, 185(5), pp. 155–184.

[60] A. Nadel, V. Ryvchin (2018). “Chronological backtracking”. In: Proceedings of 2018, pp.111-121. Springer, 2018.

[61] C. Luo, K. Su and S. Cai (2012).” Improving local search for random 3-SAT using quantitative configuration checking”. In: Proceedings of ECAI 2012, pp. 570–575.

[62] H. Fu, J. Liu and Y. Xu (2020). “Focused Random Walk with Probability Distribution for SAT with Long Clauses”. Applied intelligence. Accepted for publication.