Emphasis on the Flipping Variable: Towards Effective Local Search for Hard Random Satisfiability

Huimin Fu[a], Yang Xu[a, b,*], Guanfeng Wu[a, b,*], Jun Liu [c], Shuwei Chen[a, b] and Xingxing He[a, b]

[a]*National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University, Chengdu 610031, China*
[b]*School of Mathematics, Southwest Jiaotong University, Chengdu 610031, China*
[c]*School of Computing, Ulster University, Northern Ireland, UK*

Abstract

Uniform random satisfiability (URS) and hard random satisfiability (HRS) are two important benchmarks for algorithms that solve Boolean satisfiability problems, i.e., SAT solvers, especially for random SAT solvers. Recently, the stochastic local search (SLS) algorithms have made major breakthroughs in URS, resulting in several new 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 a new SLS algorithm, named EPEFV for HRS, which employs the extended framework of ProbSAT, and adds a new heuristic method that emphasizes the role of flipping variable, called *EFV*. *EFV* focuses on the flipping variables and is based on three components: 1) A new clause weighting scheme focusing on the flipping variable, which is based on a new clause property called *UnsatT*. By applying this new weighting scheme and a biased random walk, we design a new clause selection mechanism. 2) Design a new scoring function named U_v by combining a novel variable property *vUnsatT* based on the flipping variable with the commonly used property *score*.3) A new tie-breaking strategy in the variable selection mechanism based on the new scoring function U_v. Extensive experimental results demonstrate that EPEFV can not only greatly outperforms the state-of-the-art SLS algorithms as well as complete solver competitors on HRS instances, but also can effectively solve URS instances with long clauses. On the contrary, the most advanced SLS solvers, however, can only effectively solve URS instances, while the most advanced complete solvers can only effectively solve HRS instances. At present, no solver can effectively solve both HRS and URS at the same time, which means that the EPEFV can be regarded as the state-of-the-art SLS solver for both HRS instances and URS instances with long clauses. Finally, further empirical analysis confirms the effectiveness of each mechanism underlying the *EFV* heuristic on HRS instances.

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 [8]. The SAT problem is fundamental in solving many practical problems [12] in combinatorial optimization, statistical physics, and circuit verification. Given a formula in conjunctive normal form (CNF), the SAT problem for this formula is to answer whether it is satisfiable or not. If it is satisfiable, then a satisfying assignment should be found; otherwise, it is required to prove that there exists no satisfying assignment. During the last two decades, a significant amount of efforts has been devoted to the study of randomly generated satisfiability instances (random SAT) and the performance of different algorithms on them.

* The corresponding author.
E-mail: xuyang@home.swjtu.edu.cn (Y. Xu) and wgf1024@swjtu.edu.cn (F. Wu);

SAT algorithms can be mainly categorized into two classes: complete algorithms and incomplete algorithms [18]. Complete algorithms, including conflict-driven clause learning (CDCL) [33, 34, 44], are able to correctly judge a given formula to be satisfiable or unsatisfiable, but as the scale of the problem increases, the complexity of the solution space grows exponentially. Incomplete algorithms cannot determine that a given formula is unsatisfiable, but they are usually surprisingly effective in finding solutions of satisfiable instances for random SAT (RS) instances [24]. Especially, stochastic local search (SLS) algorithms among the incomplete algorithms are the most actively developing approach [40]. The readers are referred to [3] for other kinds of incomplete algorithms.

SLS algorithms for SAT start by randomly generating a truth assignment of the variables of a given formula. Then they explore the search space to minimize the number of falsified clauses. To do this, they iteratively adapt some heuristics to select a variable to be flipped until they seek out a solution or timeout. SLS algorithms are often evaluated on RS benchmarks [1], including uniform random *k*-SAT (URS) benchmarks and hard random SAT (HRS) benchmarks. We adopt the name of "hard random SAT" from [24] and the SAT competitions [8, 9], which are random instances generated by planting a solution using an approach named "clause distribution control" (HRS) [10]. So, they are essentially HRS-based random instances. The word "hard" is just stating that such instances are hard to solve for existing local search algorithms. Both URS and HRS have a large variety of instances to test the robustness of algorithms. Moreover, the performance of algorithms is usually stable on RS instances. Thus, we can easily recognize good heuristics by testing SLS algorithms on RS instances. The heuristics used by SLS solvers to solve RS problems are also useful for solving application problems (large-scale application problems can be transformed into SAT problems and then solved by the SAT solvers), e.g., computing theory [42], core graphs [29], gene regulatory networks [19], automated verification [45].

Among URS instances, the famous SLS algorithm WalkSAT could solve uniform random 3-SAT instances with one million variables two decades ago [23, 30]. The FrwCB solves uniform random 3-SAT instances at ratio 4.2 with millions of variables within 2-3 hours [16]. Recently, significant breakthroughs have been achieved on SLS algorithms for solving URS, resulting in the current SLS algorithms, e.g., ProbSAT [5], ProbSAT's variant YalSAT [11], ProbSAT's improvement Dimetheus [21] as well as Score$_2$SAT [17]. The ProbSAT algorithm achieves great progress in solving URS instances with a clause-to-variable ratio at the solubility phase transition, and these instances are the most difficult among RS instances [27]. ProbSAT utilizes the probability distribution strategy for variable selection [6], which has been successfully applied to URS problems. YalSAT adopts the restart strategy based on a reluctant doubling scheme (Luby) on the basis of ProbSAT and shows great success on solving URS. Dimetheus adopts preprocessing and bias-based decimation on the basis of ProbSAT and is currently the best SLS algorithm for solving huge URS instances whose ratios are not close to phase transition. In addition, it solves all uniform random 3-SAT instances at clause-to-variable less than 4.267 (r<4.267) with millions of variables within 5000 seconds according to the recent results of SAT Competitions[1].. Score$_2$SAT algorithm is developed based on the configuration checking (CC) strategy [35] and shows good performance on URS instances. However, uniform random *k*-SAT with long clauses (k>3) remains challenging for SLS solvers.

In addition, HRS instances are even harder to solve than URS instances at the solubility phase transition for SLS solvers [45]. Note that all the HRS instances counts 65% of the random benchmark in the random track of SAT Competition in 2018[2], indicating that the importance of HRS instances has

[1]http://www.satcompetition.org/

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

been highly recognized by the SAT community. As can be seen from the competition results of the random track of SAT Competitions in 2017 and 2018, although the number of variables in HRS instances is usually smaller than that in URS, the success rates of ProbSAT, Yalsat, Demetheus, and Score$_2$SAT in solving HRS lags far behind compared with their success rates in solving URS. Thus, the performance of current SLS solvers on solving HRS needs to be further improved, along with the above fact that URS with long clauses remains challenging for SLS solvers, which motivates us to design a more effective and efficient SLS algorithm for solving more classes of HRS problems as well as URS problems with long clauses.

SLS algorithms explore the search space aiming to minimize the number of unsatisfied clauses by some heuristics whose goal is to select a flipping variable. Thus, flipping variables play a rather important role in the search process. In this work, we present a new SLS algorithm named EPEFV (Extended Probability strategy with Emphasis on the Flipping Variable) to achieve the above objective. EPEFV employs an extended framework of ProbSAT incorporated with a new heuristic called *EFV*, with emphasis on the flipping variable. The *EFV* heuristic has two components: a clause selection mechanism based on the weighting scheme and the biased random walk, and a variable selection mechanism based on a novel scoring function. The main contributions are summarized as follows.

Firstly, we identify an efficient SLS algorithm as a basis for solving RS. We have chosen ProbSAT as a basis finally instead of Dimetheus, although ProbSAT shows worse performance than Dimetheus on solving URS and has similar performance to Dimetheus on solving HRS according to the results of the random track of SAT Competition in 2018. The new SLS algorithm is proposed by either replacing the existing strategies in ProbSAT or adding new strategies to ProbSAT, especially on both clause selection scheme and variable selection scheme as detailed below. It is worth noting that based on the experimental results in Section 7 and Section 8, the performance of the proposed SLS algorithm is better than Dimetheus.

Secondly, we introduce a new clause property focusing on the flipping variable called *UnsatT*, which measures the number of steps at which a clause was unsatisfied while containing the flipping variable up to a certain step (containing the flipping variable is necessary because some unsatisfied clauses do not contain the flipping variable in a certain step, that is also the main reason behind the term "emphasis on the flipping variable"). General clause weights are updated according to whether a clause is satisfied or unsatisfied by flipping a variable [17, 36], while it is achieved in *UnsatT* based on only whether an unsatisfied clause contains the flipping variable, distinguishing itself from the existing clause weighting functions. Based on the *UnsatT* property, we develop a new clause weighting scheme named UT, and define a new type of clause named HSC-UT (hard satisfiable clauses based on UT). Then, a new clause selection mechanism is proposed based on the HSC-UT clauses, the UT scheme, and the updated biased random walk strategy.

Thirdly, we introduce a new variable property emphasizing the flipping variable called *vUnsatT*, which measures the number of times at which a variable appeared in those unsatisfied clauses while containing the flipping variable up to a certain step, distinguishing itself from previous variable properties. In this work, we design a new tie-breaking strategy based on a scoring function called U_v, which is a linear combination of *score* and *vUnsatT*.

Finally, by adopting the clause selection mechanism with emphasis on the flipping variable and integrating it with the proposed variable selection mechanism with emphasis on the flipping variable, we obtain a new *EFV* heuristic. The two mechanisms underlying the heuristic *EFV* form the key components of the EPEFV algorithm.

To evaluate the efficiency and the robustness of the EPEFV algorithm, we compare the performance of EPEFV on the extensive HRS benchmarks against ProbSAT, YalSAT, Dimetheus, Score$_2$SAT as well as several complete algorithms, i.e., MapleLCMDistChronoBT [46], Cadical [28], gluHack [44], SparrowToRiss [7]. Experimental results clearly show that EPEFV outperforms all competitors, and thus establishes a new state-of-the-art SLS algorithm for solving HRS. Besides, we compare the EPEFV on URS benchmarks against ProbSAT, YalSAT, Dimetheus, and Score$_2$SAT. Experiments show that EPEFV significantly outperforms these SLS solvers on URS benchmarks with long clauses. Finally, we perform more empirical evaluations to analyze the effectiveness of the *EFV* heuristic and demonstrate its contribution to the performance of EPEFV on HRS benchmarks, as well as the influence of different clause weighting schemes on EPEFV.

The remainder of the paper is organized as follows. In Section 2, we provide some preliminary definitions and notations. Section 3 presents a brief overview of the ProbSAT algorithm. In Section 4, we introduce a new clause selection mechanism with an emphasis on the flipping variable. Section 5 describes the new variable selection mechanism with an emphasis on the flipping variable. In Section 6, we present the EPEFV algorithm and describe it in detail. Section 7 conducts extensive experiments on HRS benchmarks to present the effectiveness and efficiency of the EPEFV, and Section 8 conducts large experiments on URS benchmarks to present the generality and applicability of the EPEFV. In Section 9, we empirically analyze the relationship of *UnsatT* and *vUnsatT* and the effectiveness of each component underlying the *EFV* heuristic on HRS benchmark, then list the main differences between EPEFV and ProbSAT as well as the major differences between UT and popular clause weighting schemes. Section 10 concludes this paper and provides some future research directions.

2. Preliminaries

A SAT instance F is defined by a pair $F=(X, C)$ such that $var(F)=\{v_1, v_2, \ldots, v_n\}$ is a set of n Boolean variables (their values belong to the set $\{true, false\}$) and $C=\{c_1, c_2, \ldots, c_m\}$ is a set of m clauses. A clause $c_i \in C$ is a disjunction of literals and a literal is either a variable v_i or its negation $\neg v_i$. For a formula F, we use $r = m/n$ to denote its clause-to-variable ratio. A formula $F= c_1 \wedge c_2 \wedge \ldots \wedge c_m$ is a conjunction of clauses, i.e., a CNF. A uniform random k-SAT instance is such that each clause contains exactly k distinct non-complementary literals. A satisfying assignment α for a CNF formula F is an assignment to its variables such that the formula is evaluated to be true. Given a CNF formula, the SAT problem will find an assignment that satisfies all the clauses of F.

SLS algorithms for SAT typically start by randomly assigning to every variable appearing in a given formula a value of either true or false; then, in each subsequent search step, a variable is selected to flip its truth assignment from true to false or vice versa. In SLS algorithms, for a variable v and an assignment α, *score* (v, α) measures the increase in the number of satisfied clauses by flipping the assigned value of v in our algorithm, and *break* (v, α) is the number of satisfied clauses that become unsatisfied by flipping the assigned value of v.

The satisfiable uniform random k-SAT generator generates satisfiable instances with planted solutions according to the q-hidden model [2]. The URS benchmarks are generated for two different sizes: medium and huge [5]. The medium-sized benchmarks are such instances with various variables and r equals to the phase-transition ratio. The huge-sized benchmarks are such instances with a few million clauses and with the ratio from far from the phase-transition ratio to relatively close and are as large as some of the application benchmarks. Especially, most (nearly 66.6% of) URS instances in the

benchmark of the random SAT track in SAT Competition 2018 are huge ones. URS instances have been added to the random track of SAT Competition since 2004.

The hard random satisfiability (HRS) is particularly interesting because it turns out to be one of the hardest for all SLS solvers [8, 9]. The satisfiable hard random instances are generated by planting a solution using the **C**lause **D**istribution **C**ontrol approach [10], and thus these instances are named as HRS-based instances. The word "hard" is just saying such instances are hard for existing local search algorithms to solve. The author [8] has indicated that HRS problems have some potential in applications (e.g., generating HRS problems with a planted solution can be used in cryptography). HRS was added for the first time to the random track of SAT Competition in 2016 to evaluate and improve SAT solvers, especially for SLS solvers. As witnessed in SAT competitions since 2016, 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 is 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.

3. ProbSAT Algorithm Overview

In this section, we briefly review ProbSAT algorithm [5], which serves as the basis of the proposed SLS algorithm. ProbSAT algorithm has wide influence among current SLS algorithms and attracted increasing interest for solving RS benchmarks in the last few years.

ProbSAT uses only the *break* values of a variable in a probability function $f(v,a)$ including a polynomial or exponential shape as listed below.

$$f(v,a) = (0.9 + break(v,\alpha))^{cb_1} \text{ or } f(v,a) = (cb_2)^{-break(v,a)},$$

where cb_1 and cb_2 are decimal parameters.

Note that ProbSAT algorithm is designed for solving HRS. The pseudo-code of ProbSAT is described in Algorithm 1 and can be found in the literature [6].

Algorithm 1 The ProbSAT Algorithm

Input: CNF-formula *F*, *MaxTries*, *MaxSteps*
Output: A satisfying assignment α of *F*, or *Unknown*
1: **for** *i:* = 1 to *MaxTries* **do**
2: α := a randomly generated truth assignment;
3: **for** *j:*= 1 to *MaxSteps* **do**
4: **if** α *satisfies F* **then return** α;
5: *C* := an unsatisfied clause chosen at random;
6: *v* := *x*∈ *C* selected with probability $\frac{f(x,\alpha)}{\sum_{z\in C} f(z,\alpha)}$;
7: α := α with *v* flipped;
8: **return** *Unknown*;

Initially, ProbSAT algorithm performs the first loop until it finds a satisfying assignment or reaches the first limited steps denoted by *MaxTries* (*MaxTries* = 10^{19}). Then ProbSAT algorithm generates a complete assignment α randomly as the initial assignment (line 2 in Algorithm 1). Then ProbSAT algorithm starts the second loop until a satisfying solution is found or reaches the second limited steps denoted by *MaxSteps* (*MaxSteps* = 10^{19}). During the search process, ProbSAT algorithm selects an unsatisfied clause randomly (line 5 in Algorithm 1), and then for solving 3-SAT instances, ProbSAT

chooses the polynomial function; otherwise, ProbSAT chooses the exponential function. ProbSAT tries to select a flipping variable based on probability (line 6 in Algorithm1) to be flipped (line 7 in Algorithm 1). Finally, once the search process terminates, the ProbSAT reports α as the solution; otherwise, ProbSAT reports UNKNOWN.

On the one hand, ProbSAT algorithm explores the search space to minimize the number of unsatisfied clauses, and to do this, it is natural for the ProbSAT algorithm to select a variable to be flipped, and thus each flipping variable is a rather important feature in the search process. The variable selection of ProbSAT mainly depends on two factors: clause selection strategy and variable selection strategy. Therefore, the heuristic emphasizing the flipping variable (named *EFV*) is suggested in the EHC heuristic [39]. However, EHC may not be suitable for the HRS. Since there are no hard clauses and soft clauses (All clauses in a weighted partial CNF formula are divided into hard ones and soft ones, and each soft clause is associated with a positive integer as its weight) [39] in HRS and the flipping variable decides the direction of the search, it is reasonable for us to employ a heuristic emphasizing the flipping variable to solve HRS. To further improve SLS algorithms for HRS, we focus on proposing two new selection heuristics with emphasis on the flipping variable, which is detailed in subsequent Sections 4 and 5 respectively.

4. Clause Selection Mechanism with Emphasis on the Flipping Variable

The strategy of picking an unsatisfied clause is known to be successful for general SAT solving [5]. Indeed, the condition that the selected clause is unsatisfied is necessary, as selecting a satisfied clause may lead to a local optimum [27]. As can be clearly seen from Algorithm 1, the ProbSAT algorithm does not distinguish unsatisfied clauses in each step. In our opinion, this is a disadvantage of ProbSAT when it is applied to RS solving. Because each unsatisfied clause varies in how easily it can be converted from being unsatisfied to satisfied, selecting from the unsatisfied clauses with equal probability does not provide sufficient guidance for SLS algorithms, especially for HRS instances. The number of times an unsatisfied clause contains the flipped variable is an indication of how difficult it satisfies the clause. We take this observation as the basis for a new clause weighting scheme that distinguishes between unsatisfied clauses.

To improve the performance of ProbSAT on solving SAT, we develop a new clause selection mechanism, which separates unsatisfied clauses in each step. The clause selection mechanism includes three components as detailed in the subsequent sections: a new clause weighting scheme (named UT), hard satisfiable clauses based on UT (HSC-UT), and a biased random walk strategy.

4.1 A New Clause Weighting Scheme Focusing on the Flipping Variable

Clause weighting schemes have been used prominently in SLS algorithms for solving SAT [4, 14, 20, 35], such as DLM [50], PAWS [49], and SAPS [26]. Although these mainstream clause weighting SLS algorithms differ in the manner of how 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. These mainstream clause weighting schemes are simply categorizing clauses into unsatisfied ones and satisfied ones, which are also witnessed by mainstream SLS solvers such as Sparrow [4], DCCASat [35], and Score$_2$SAT [17]. Moreover, as can be seen from the competition results of the random track of SAT Competitions 2017 and 2018, these SLS solvers, including Score$_2$SAT, DCCASat, Sparrow, ProbSAT, and Dimetheus [21], lost their power and effectiveness on solving HRS. Thus, these clause weighting schemes are not

informative enough to guide the SLSs for HRS instances. This motivates us to design a new clause weighting scheme that could distinguish unsatisfied clauses in each step in a more effective way.

Accordingly, we consider a new clause property named *UnsatT*, which is the number of times that a clause is unsatisfied and contains the flipping variable up to a certain step. *UnsatT* is formally defined as below:

Definition 1. *For a clause c, in each step s, UnsatT(c, s) is the number of steps at which a clause was unsatisfied while containing the flipping variable up to step s.*

In this sense, *UnsatT* can be regarded as the generalization of the property of clauses, i.e., *UnsatT* can be widely used to improve the performance of the SLS algorithm like *break* property [4]. Intuitively, clauses with larger *UnsatT* values are harder to keep satisfied in the search process. Thus, it is beneficial for SLS algorithms to satisfy these clauses at the first instance.

To assign higher priority on clauses with larger *UnsatT* in clause weights, a new clause weighting scheme based on *UnsatT* is proposed, which only works for the unsatisfied clauses which contain the flipping variable during the search steps. The new clause weighting scheme, denoted as UT, works as follows:

- At the beginning of the SLS algorithm, after an initial assignment α is generated, for a clause c, if c is unsatisfied under α, the weight of c (i.e., $UnsatT(c, 0)$) is set to be 1; otherwise, $UnsatT(c, 0) = 0$;
- In search step s, if c is unsatisfied and contains the flipping variable, then $UnsatT(c, s) = UnsatT(c, s\text{-}1) + 1$;
- Otherwise, $UnsatT(c, s) = UnsatT(c, s\text{-}1)$.

Accordingly, the proposed algorithm based on UT only checks the unsatisfied clauses containing the flipping variable rather than checking all clauses, and thus saving the computation time. Here we utilize UT to guide the clause selection, distinguishing itself from previous clause weighting schemes in SLS algorithms on picking a variable [4, 14, 35].

To pick a clause based on UT, inspired by HCSCCD (Hard Clauses' States based Configuration Changed and Decreasing) variables [39] in SLS algorithms, we introduce the notions of HSC-UT (hard satisfiable clause based on UT) and ESC-UT (easily satisfiable clauses based on UT). The formal definitions of HSC-UT and ESC-UT are given as follows:

Definition 2. *For a clause c, in search step s, and given a positive integer parameter β, c is called an HSC-UT in step s if c is unsatisfied and $UnsatT(c, s) \geq \beta$.*

Definition 3. *For a clause c, in search step s, and given a positive integer parameter β, c is called an HSC-UT in step s if c is unsatisfied and $UnsatT(c, s) < \beta$.*

In this work, when the SLS algorithm searches to step s, we use the notation HSC-UT(s, β) to denote the set of all HSCs-UT in step s and ESC-UT(s, β) to denote the set of all ESCs-UT in step s respectively for the given β. In search step s, the union of HSC-UT(s, β) and ESC-UT(s, β) is the set of all unsatisfied clauses in search step s for the given β. HSCs-UT is regarded as good candidates for clause selection, especially when solving HRS problems.

4.2 The Biased Random Walk Strategy

An important component of ProbSAT algorithm is the standard random walk (line 6 in Algorithm 1). The standard random walk has been utilized prominently in SLS algorithms, including WalkSAT [23], ProbSAT, YalSAT [11], and Dimetheus [21]. However, these SLS solvers are ineffective on

solving HRS instances, which was also illustrated by recent SAT Competitions. Thus, the standard random walk may not be suitable for HRS instances. As discussed in Section 4.1, HSCs-UT is assigned higher priority to be satisfied for HRS in the proposed algorithm, and thus it is reasonable for us to adopt a biased random walk strategy inspired by the idea from [39], which is utilized to select a clause and described as follows:

- At each step s, when a biased random walk is called, if $HSCs$ (s, β) is not empty, then an HSC-UT clause is selected randomly;
- Otherwise, an ESC-UT clause is selected randomly.

In summary, in the proposed new SLS algorithm, we replace the standard random walk component (lines 6 in Algorithm 1) with the above described biased random walk component. As the biased random walk component would select a clause randomly from either HSC-UT(s, β) or ESC-UT(s, β) in any step s for a given β, thus a key point to efficiently implement biased random walk is to maintain two clause sets during the search process: the set of current HSCs-UT and the set of current ESCs-UT. Following the above clause selection scheme, the algorithm will select a variable in the chosen clause, which is based on a new variable selection mechanism introduced in the subsequent section.

5. Variable Selection Heuristic with Emphasis on the Flipping Variable

In this section, we introduce a new variable selection heuristic with emphasis on the flipping variable, named EFV, which is composed of three components: probabilistic variable selection, a new variable property (namely *vUnsatT*), and a tie-breaking strategy based on a new scoring function (called U_v), as detailed below.

5.1 Probabilistic Variable Selection

The *break* values count the number of clauses that become unsatisfied by flipping a given variable, and thus it is natural to give priority to variables that have the smallest *break* value. Given a selected clause, we adopt the probability function f (described in Section 3) used in ProbSAT [5] to probabilistically select variables that have smaller *break* values. However, using only probability-based on f may result in the same variable being selected in consecutive steps. To avoid this, a new variable property (namely *vUnsatT*) is introduced to design a new scoring function (called U_v), and then we employ a tie-breaking strategy based on U_v.

5.2 A New Variable Property- *vUnsatT*

In SLS algorithms, the method for selecting the variable to be flipped in each step is usually guided by a scoring function. A scoring function can be a simple variable property or any mathematical expression with one or more properties. The *score* property minimizes the number of currently unsatisfied clauses in CNF. However, as indicated in Section 4.1, simply focusing on the number of satisfied clauses and unsatisfied clauses is not informative enough to guide the SLS algorithm, especially for HRS.

Therefore, we consider the number of times each variable appears in those unsatisfied clauses containing the flipping variable in each search step. This measurement of a variable can be regarded as a new variable property, distinguishing itself from previous variable properties and is defined formally as below:

Definition 4. *For a variable v, in the search step s, vUnsatT(v, s) is the number of steps at which a variable v appeared in those unsatisfied clauses while containing the flipping variable up to step s.*

In this sense, *vUnsatT* can be regarded as the generalization of the property of variables, i.e., *vUnsatT* can also be widely used to improve the performance of the SLS algorithm like *score* property [16]. Intuitively, clauses containing variables with larger *vUnsatT* are harder to keep being satisfied in the search process, and we use *vUnsatT* to guide the variable selection. The update process of *vUnsatT* is given below.

- After an initial assignment α is generated, for a variable v, if v appears in t unsatisfied clauses under α, then $vUnsatT(v, 0)=t$; otherwise, if v does not appear in any unsatisfied clauses, $vUnsatT(v, 0) =0$;
- In search step s, once a flipping variable has been selected, let $m_v(s)$ denote the number of times that v appears in those unsatisfied clauses while containing the flipped variable in step s. If $m_v(s)\neq 0$, then $vUnsatT(v, s) = vUnsatT(v, s\text{-}1) + m_v(s)$;
- Otherwise, if $m_v(s)=0$, $vUnsatT(c, s) = vUnsatT(c, s\text{-}1)$.

Remarks: when the variable property *vUnsatT* is updated, the algorithm only checks the variables that appear in the unsatisfied clauses while containing the flipped variable in step *s* rather than checking all variables and thus saves the computation time. In addition, it can be seen from the function *vUnsatT* (*v*, *s*) that, as the step *s* increases, *vUnsatT* (*v*, *s*) is monotonically increasing with respect to variable *v*.

5.3 Scoring Function of Variable – U_v

The *score* property tends to decrease the number of unsatisfied clauses in the greedy mode. The *vUnsatT* property can be used as a heuristic for greedy search as its use tends to reduce the set of HSCs-UT by flipping a variable of an HSC-UT.

When deciding the priority of candidate variables to be selected, although *score* property is more important than *vUnsatT* property, in some cases, *vUnsatT* should be allowed to overwrite the priorities. Given a CNF formula *F*, the maximum value of *score* for all variables does not exceed the number of clauses during the search process. However, as the search process progresses, the *vUnsatT* value of each variable increases rapidly, and the more step the search proceeds, the larger the value of *vUnsatT* will be.

To combine *score* and *vUnsatT* in a greedy search, we introduce a new scoring function that is a linear combination of *score* and *vUnsatT*, inspired by the concept of comprehensive *score* [16]. The new scoring function, named U_v (because it is utilized to break the tie of variable as detailed in the subsequent section), is defined as follows:

Definition 5. *For a variable v, in search step s, when the assignment is α, the scoring function, denoted as U_v, is defined as:*

$$U_v(v, s, \alpha) = score(v, \alpha) + vUnsatT(v, s)/\gamma,$$

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

U_v is so simple that it can be computed with little overhead, and the parameter γ can be easily tuned. Moreover, its simplicity allows its potential usage in solving many SAT instances and perhaps other combinatorial search problems.

5.4 The New Tie-breaking Strategy

In the above sections, some measures or evaluations of a variable property have been discussed and form the basis for the new variable selection heuristic. Currently, there are two most popular variable selection strategies for solving RS: probability function strategy [6] and CC strategy [14].

In ProbSAT, it may result in selecting the same variable in consecutive steps by adopting only the probability function f to pick a variable to be flipped, so that it causes useless work in consecutive steps. Therefore, based on one idea of CC strategy [13, 38], it is expected to remember each variable's circumstance information and prevents a variable from being flipped if its circumstance has not been changed since its last flip, which has been proved to be effective in the SLS algorithm for solving URS instances [17, 35]. CC strategy is technically hard to track and realized by selecting a variable from all variables in a SAT formula, while our algorithm selects a variable from an unsatisfied clause chosen by the biased random walk, i.e., CC-based SLS solvers [13, 14, 16, 17, 35] have only the variable selection, while our algorithm needs to select a clause and then select a variable from this selected clause. Thus, the CC strategy may not be suitable for our algorithm, and it is reasonable for us to employ a tie-breaking strategy that avoids selecting the same variable in consecutive steps.

The new tie-breaking scheme in the variable selection heuristic is described as follows:

In our algorithm, if the variable selected by probability at step s is the same as the variable flipped in step s-1, a different variable with the greatest U_v value will be selected instead (further ties are broken by picking the variable that comes earliest in the clause).

The proposed tie-breaking strategy is inspired by the idea in the literature [16], but they are essentially different from each other due to the fact that the latter may not be suitable for HRS. The main difference in our proposal lies in that a variable is mainly selected based on the probability function, and there is no need to select one from all those variables with the same minimum break value in the selected clause.

In brief, in the chosen clause based on the EFV heuristic, the new variable selection mechanism is achieved by selecting the variables based on the probability function f; once ties occur, a new tie-breaking strategy breaks ties of variables and selects a variable by preferring the variable with the greatest U_v value, which is a linear combination of *score* and *vUnsatT*.

6. The EPEFV Algorithm

Based on the ProbSAT framework and two selection mechanisms underlying the EFV heuristic described in Sections 4 and 5, we present a new SLS algorithm for solving random SAT, named EPEFV (**E**xtended Probability strategy with **E**mphasis on the **F**lipping **V**ariable) in this section. The pseudo-code of the EPEFV algorithm is outlined in Algorithm 2. We describe it in detail as follows.

Algorithm 2 The *EPEFV* Algorithm

Input: CNF-formula F, *MaxTries*, *MaxSteps*, γ, β
Output: A satisfying assignment α of F, or *Unknown*
1: **for** *i:* = 1 to *MaxTries* **do**
2: α := a randomly generated truth assignment;
3: *bestVar* := null;
4: **for** *j:*= 1 to *MaxSteps* **do**
5: **if** α *satisfies* F **then return** α;
6: **if** *HSC-UT*(j, β) *is not empty* **then**
7: C := a clause randomly chosen from *HSC-UT*(j, β);
8: **else** C := a clause randomly chosen from *ESC-UT*(j, β);
9: $v := x \in C$ selected with probability $\frac{f(x,\alpha)}{\sum_{z \in C} f(z,\alpha)}$;
10: **if** v :== *bestVar* **then**
11: *bestVar* := $x \in C, x \neq v,$ with the greatest $U_v(x, s, \alpha)$ in C, breaking ties with the smallest order;
12: **else** *bestVar* := v;
13: $\alpha := \alpha$ with *bestVar* flipped;
14: update *UT and vUnsatT*;
15: **return** *Unknown*;

Initially, EPEFV performs the first loop until it finds a satisfying assignment or reaches the first limited steps denoted by *MaxTries*. Then EPEFV generates a complete assignment α at random as the initial solution. *bestVar* is used to record which variable was flipped in the last step (line 3). Then it executes the second loop until a solution is found or reaches the second limited steps denoted by *MaxSteps*. The value of *MaxSteps* and *MaxTries* are set the same as those in ProbSAT in Algorithm 1. In each search step, EPEFV picks a variable to be flipped. EPEFV performs the biased random walk component as detailed in Section 4.2 (lines 6-8 in Algorithm 2): if there exists any HSC-UT in any step j, a clause is picked randomly from HSC-UT(j, β); otherwise, a clause is picked randomly from ESC-UT(j, β). The algorithm then picks a variable according to the probability based on f and the new tie-breaking strategy as detailed in Section 5.3 (lines 9-12 in Algorithm 2): EPEFV picks first a variable by the probability based on f (line 9 in Algorithm 2); if the variable is the same as the last flipped variable (*bestVar*), EPEFV selects a variable by preferring the variable with the greatest U_v value (line 11). After the variable is selected, the EPEFV flips the selected variable (line 13 in Algorithm 2) and updates the clause weights based on the weighting scheme UT as detailed in Section 4.1 and also updates the *vUnsatT* as detailed in Section 5.1 (line 14 in Algorithm 2), then the EPEFV algorithm starts the next search step.

Finally, when the search terminates, if α satisfies all clauses of F, EPEFV outputs α as the solution; otherwise, EPEFV reports *UNKNOWN*.

7. Experimental Evaluations on HRS Benchmarks

In this section, we first introduce the benchmark instance sets, the competitors, and the experimental setup utilized in our experiments. Then we compare EPEFV with state-of-the-art SLS solvers and complete solvers on all HRS testing benchmarks.

7.1 The Benchmarks

To make the experimental evaluation more comprehensive, apart from the existing HRS benchmarks taken from the latest SAT competitions, additional HRS instances are generated according

to the hard random SAT formula tool [9], which randomly generates SAT instances significantly harder than URS of the same size. We adopt the following benchmarks for HRS for testing purposes as well.

1) **4.3HRS Comp17:** all HRS instances with r=4.3 from SAT Competition 2017(400≤n≤540, 40 instances, 5 for each size)
2) **4.3HRS Testing**: HRS instances generated randomly by the hard random SAT formula tool (r=4.3, 600≤n≤1000, 1000 instances, 200 for each size)
3) **5.206HRS Comp17:** all HRS instances with r=5.206 from SAT Competition 2017 (400≤n ≤540, 40 instances, 5 for each size)
4) **5.206HRS Testing**: HRS instances generated randomly by the hard random SAT formula tool (r=5.206, 600≤n≤1000, 1000 instances, 200 for each size)
5) **5.5HRS Comp17:** all HRS instances with r=5.5 from SAT Competition 2017(400≤n≤540, 40 instances, 5 for each size)
6) **5.5HRS Testing**: HRS instances generated randomly by the hard random SAT formula tool (r=5.5, 600≤n≤1000, 1000 instances, 200 for each size)
7) **5.699HRS Training**: HRS instances generated randomly by the hard random SAT formula tool (r=5.699, 200≤n

   ≤1000, 45 instances, 5 for each size)
8) **SAT Comp18**: all HRS instances from SAT Competition 2018 (r=4.3, r=5.206, r=5.5, 200≤n≤400, 165 instances, 55 for each ratio). Note that these HRS instances occupy 65% of random benchmark in SAT Competition 2018, indicating that the importance of HRS instances has been highly recognized by the SAT community.
9) **5.699HRS Testing**: HRS instances generated randomly by the hard random SAT formula tool (r=5.699, 200≤n≤1000, 900 instances, 100 for each size)

7.2 State-of-the-art SLS and Complete Competitors

First of all, the proposed EPEFV algorithm is compared against four SLS solvers including Score$_2$SAT [17], Yalsat [11], ProbSAT [6] and Dimetheus [22] as well as four complete solvers including Cadical [28], Maple_LCM_Dist_ ChronoBT [46], gluHack [44], and SparrowToRiss [7].

Note that ProbSAT is the basic framework of EPEFV and is also the basic framework of Dimetheus and YalSAT. Yalsat won the random track of SAT competition in 2017 (RTS2017). As reported in the results of 2017[3], Yalsat significantly outperforms an efficient complete algorithm tch_glucose3. Dimetheus is the best SLS solver for URS instances. Dimetheus won the RSC2014[4] and RSC2016[5] and also first place among the SLS algorithms in RSC2018[6]. According to the results of the RTS2016 and RTS2018, Dimetheus performs much better than DCCAlm [37], CSCCSat [36], YalSAT, and ProbSAT. In our experiments, we use the versions of Dimetheus, ProbSAT, and Yalsat submitted to RTS2018. The Score$_2$SAT algorithm is the best SLS solver for HRS and won third place in RTS2017.

[3]https://baldur.iti.kit.edu/sat-competition-2017/results/random.csv.
[4]www.satcompetition.org/2014 /results.shtml.
[5]https://baldur.iti.kit.edu/sat-competition-2016/index.php?cat=results.
[6]http://sat2018.forsyte.tuwien. ac.at/index.php?cat=results.

In our experiments, the Score$_2$SAT is downloaded from the webpage of SAT Competition 2017.

SparrowToRiss (denoted by STR for convenience) is a complex solver among preprocessor CP3 [41], Sparrow [4], and Riss [7], and is the best complete solver for HRS, and won the RTS2018. As reported in the results of the RTS2018, STR shows superiority over Dimetheus. The algorithm gluHack is an efficient complete solver and won the silver of RTS2018. For STR and gluHack, we use the binary from SAT Competition 2018. MapleLCMDistChronoBT (denoted by MBT) won the main track of 2018 SAT Competition (MTS2018). In our experiments, the binary of MBT is the one submitted to MTS2018. Cadical is the best complete solver for application instances and solved the most instances in MTS2019. In our experiments, the source code of Cadical can be downloaded online[7].

7.3 Experimental Preliminaries

EPEFV is implemented in C. We tuned the β and γ parameters of EPEFV according to our experience. Accordingly, the optimal parameters are summarized in Table 1. For c_1 and c_2, we utilize the default parameter setting tuned in the literature [7].

Table 1

Parameters settings of β and γ for EPEFV on solving HRS benchmarks.

Variable n	$r\leq4.3$		$4.3<r<5.5$		$r\geq5.5$	
	β	γ	β	γ	β	γ
$n\leq400$	0	1000	215	321	2380	911
$400<n\leq600$						961
$n>600$	3	1022		1212		1205

All experiments are carried out on the machine under a 64-bit Ubuntu Linux Operation System, using 2 cores of Intel(R) Core (TM) i7-6700M 3.4 GHz CPU and 16 GB RAM. Each run that terminates in finding a satisfying assignment within the cutoff time is successful. The cutoff time is set to 5000 seconds for the 4.3HRS Comp17 benchmark, 5.206HRS Comp17 benchmark, 5.5HRS Comp17 benchmark, and SAT Comp18 benchmark (as in SAT Competitions 2017, 2018, and 2019), and 600 seconds for the rest benchmarks (as in the literature [9]).

For the HRS instances from SAT Competition 2017, we run each solver 10 times for each instance. For the HRS instances randomly generated and the SAT Comp18, we run each solver one time for each instance, as the instances in each ratio are sufficient to test the performance of the solvers [16].

For performance metrics, we report successful runs ("suc") and the penalized average run time ("par 2") (an unsuccessful run is penalized as double cutoff time) (as in SAT Competitions). The best results for an instance class are highlighted in bold. If a solver has no successful run for a group of instances, the corresponding 'par 2' is marked with "-".

7.4 Experimental Results

In this subsection, we summarize the experimental results of EPEFV compared with its SLS and complete competitors on the testing benchmarks for HRS as introduced in Section 7.1.

Table 2 presents the comparative experimental results of EPEFV with its SLS competitors Score$_2$SAT, YalSAT, Dimetheus as well as ProbSAT and complete competitors gluHack, STR, MBT and Cadical on 4.3HRS Comp17 benchmark, 4.3HRS testing benchmark, 5.206HRS Comp17

[7]http://sat-race-2019.ciirc.cvut.cz/

benchmark, 5.206HRS Testing benchmark, 5.5HRS Comp17 benchmark, 5.5HRS Testing benchmark, SAT Comp18 benchmark and 5.699HRS testing benchmark (Detailed breakdown results are shown in **Tables 1- 5 in the Appendix**).

Table 2 shows that EPEFV is clearly the best solver on these benchmarks of HRS instances. EPEFV gives the best performance for all HRS instance classes except for 4.3HRS Comp17 and 4.3HRS testing, and especially it solves more 5.206HRS Testing, 5.5HRS Testing, and 5.699HRS testing instances than all other solvers. Given the good performance of EPEFV on 5.206HRS Testing benchmark with up to1000 variables, it is very likely that it could be able to solve larger HRS instances with r=5.206, r=5.5, and r=5.699. For 4.3HRS Comp17 and 4.3HRS testing, EPEFV solves as many instances as all SLS competitors, but par 2 is a little more than those of all SLS competitors. These experimental results confirm the good performance of EPEFV on the HRS benchmarks in SAT Competition 2018, where it also solved more HRS instances than all SLS competitors and spent less time than all complete competitors.

Table 2

Computational results on the HRS instances.

Benchmark	Score$_2$SAT		YalSAT		Dimetheus		ProbSAT		gluHack		STR		MBT		Cadical		EPEFV	
	suc	par2	suc	par2	suc	par2	suc	par2	suc	par2	suc	par2	suc	par2	suc	par2	suc	par2
4.3HRS	400		400		400		400		290		400		270		350		400	
Comp17	0.020		**0.017**		0.057		0.062		3481		0.117		3641		1625		0.023	
4.3HRS	1000		1000		1000		1000		40		1000		0		80		1000	
Testing	0.028		**0.017**		0.028		0.024		1347		0.376		-		1119		0.223	
5.206HRS	30		0		0		0		380		400		400		380		400	
Comp17	9250		-		-		-		863.2		5.709		67.64		1179		**0.039**	
5.206HRS	0		0		0		0		80		800		193		120		**1000**	
Testing	-		-		-		-		323.2		261.2		1010		1642		**0.079**	
5.5HRS	90		90		90		90		400		400		400		400		400	
Comp17	7750		7750		7750		7750		24.90		151.0		3.785		15.23		**0.662**	
5.5HRS	160		160		160		160		360		460		920		440		**1000**	
Testing	1008		1008		1008		1008		839.8		650.3		158.1		751.6		**1.298**	
SAT	100		94		79		79		165		165		165		165		165	
Comp18	3939		4349		5526		5259		5.476		45.82		4.544		21.13		0.**128**	
5.699HRS	0		0		0		0		600		520		800		559		**900**	
Testing	-		-		-		-		438.1		608.7		179.7		473.4		**0.768**	

8. Experimental Evaluations on URS Benchmarks

Section 7 is focused on a comprehensive evaluation of the performance of EPEFV compared with the SLS and complete competitors on the existing HRS benchmarks from the latest SAT competitions (2017 and 2018), along with additional HRS instances generated according to the hard random SAT formula tool [9]. To show the generality and applicability of the proposed EPEFV algorithm, additional experiments on the uniform random k-SAT (URS) benchmarks are carried out, and the results are summarized in this section. More specifically, results of extensive experiments to evaluate EPEFV on uniform k-SAT instances with long clauses are provided, now that URS with long clauses remains challenging for SLS solvers.

8.1 The Benchmarks and Experiment preliminaries

To make the experimental evaluation more comprehensive, apart from the existing URS benchmarks from the latest SAT Competitions (2016 and 2017), additional URS instances are generated

according to the URS generator[8], with additional application instances from SAT Race 2019. Specially, we adopt the following benchmarks:

1) **SAT Competition 2017**: all random *k*-SAT instances with *k*>3 from SAT Competition 2017 (120 instances, 60 for each *k*-SAT, *k*=5, 7), which vary in both size and ratio. These instances count 67% of the random URS benchmark in SAT Competition 2017 (these instances occupy 40% of all random benchmark including URS and HRS in SAT Competition 2017). The instances at *r*=21.117 vary from 200 variables to 590 variables, and with *n*=250000 vary from 16.0 ratios to 19.8 ratios for 5-SAT. The instances at *r*=87.79 vary from 90 variables to 168 variables, and with *n*=50000 vary from 55.0 ratios to 74.0 ratios for 7-SAT.
2) **5-SAT medium:** all URS instances at phase transition generated randomly according to the URS generator (*r*=21.117, *n*=200, 250, 300, 60 instances, 20 for each size)
3) **7-SAT medium:** all URS instances at phase transition generated randomly according to the URS generator (*r*=87.79, *n*=100, 110, 120, 60 instances, 20 for each size)
4) **SAT Competition 2016 medium**: all random *k*-SAT instances at phase transition with long clauses (80 instances, 40 for each *k*-SAT, *k*=5, 7). The instances at *r*=21.117 vary from 200 variables to 590 variables for 5-SAT, and the instances at *r*=87.79 vary from 90 variables to 168 variables for 7-SAT.
5) **5-SAT huge:** all URS instances whose ratios are not that close to phase transition while they have huge sizes generated randomly according to the URS generator (*r*=18.0, 18.2, 18.4, *n*=250000, 150 instances, 50 for each size)
6) **7-SAT huge:** all URS instances with huge sizes generated randomly according to the URS generator (*r*=64, 65, 66 *n*=50000, 150 instances, 50 for each size)

The parameter settings are determined according to our experience for β and γ are summarized in Table 3.

Table 3

Parameters settings of β and γ for EPEFV on solving URS benchmarks.

k-SAT	huge		medium	
	β	γ	β	γ
5-SAT	800	50	10000000	50000000
7-SAT			50000000	20000000

The computing environments for the experiments are the same as those utilized for experiments in Section 7. We perform each solver 10 runs for each instance from SAT Competitions in 2017 and 2016. For the remaining benchmarks, each solver is performed one run on each instance. Each run that terminates in finding a satisfying assignment within the cutoff time is successful. The cutoff time is set to be 5000 seconds. We report successful runs ("suc") and the penalized average run time ("par 2") (as in SAT Competitions). The best results for an instance class are highlighted in bold. If a solver has no successful run for a group of instances, the corresponding 'par 2' is marked with "-".

8.2 Experimental results

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

[8]https://sourceforge.net/projects/ksat generator/

Table 4 presents the comparative results of EPEFV with its SLS competitors Score$_2$SAT, YalSAT, Dimetheus as well as ProbSAT on SAT Competition 2017 benchmark, SAT Competition 2016 benchmark, 5-SAT medium benchmark, 7-SAT medium benchmark, 5-SAT huge benchmark, and 7-SAT huge benchmark (Detailed breakdown results are shown in **Tables 6- 10 in the Appendix**).

Since EPEFV is based on ProbSAT, we first compare these two solvers. As shown in Table 4, EPEFV solves more instances than ProbSAT on all instance classes except for the 7-SAT medium. For each benchmark, ProbSAT succeeds in 533 runs, 321 runs, 29 runs, 32 runs, 48 runs, 97 runs respectively, while EPEFV succeeds in 621 runs, 330 runs, 30 runs, 30 runs, 150 runs, 150 runs respectively.

According to Table 4, EPEFV solves a few more instances than Score$_2$SAT, YalSAT, and Dimetheus. Further observation shows that EPEFV has similar performance with all SLS competitors on 5-SAT medium and 7-SAT medium instances, and has similar performance as the best solver Dimetheus on 5-SAT huge and 7-SAT huge instances, but with less time.

Table 4

Computational results on the URS instances.

Benchmark Class	Dimetheus		ProbSAT		YalSAT		Score$_2$SAT		EPEFV	
	suc	par2	suc	par2	suc	par2	suc	par2	suc	par2
SAT Competition 2017	592	5173	533	5713	510	5890	534	6077	**621**	**5138**
5-SAT medium	31	**4895**	29	5260	31	5014	29	5360	30	5133
7-SAT medium	33	4690	32	4857	33	4857	**33**	**4622**	30	5134
SAT Competition 2016 medium	311	6383	321	6239	282	6657	311	4808	**330**	**6102**
5-SAT huge	150	**626.5**	48	7073	96	4260	50	7927	150	1552
7-SAT huge	150	**318.9**	97	3812	21	10721	91	6303	150	473.1

9. Discussions

In this section, we present a detailed discussion of the EPEFV algorithm on HRS benchmarks. First, we conduct further empirical analyses to reveal the relationship between *UnsatT* and *vUnsatT*, and present the effectiveness of each underlying component in the EFV heuristic. Then we discuss the major differences between EPEFV and ProbSAT. Finally, we discuss the main differences between the clause weighting scheme UT and the popular clause weighting schemes PAWS and SWT.

9.1 Relationship between *UnsatT* and v*UnsatT* as well as Their Impact on the Effectiveness of the EFV Mechanism

In this subsection, we illustrate the impact of *UnsatT* and *vUnsatT* on the effectiveness of the EFV heuristic through both theoretical and experimental analysis.

Intuitively, in a clause *c*, if the maximum difference between *vUnsatT* and *UnsatT* is so small that all variables in *c* are not connected to other clauses, then the *vUnsatT* property becomes ineffective because the tie-breaking strategy depends only on the *score* function of the variable. We will go deep into this intuition by analyzing the relationship between *UnsatT* and *vUnsatT* in HRS instances and demonstrating their influence on the effectiveness of the variable selection heuristic.

For a HRS formula *F*, when EPEFV performs until step *s*, for the relationship between *UnsatT* and *vUnsatT* in *F*, we have the following conclusion.

Theorem 1. *For a CNF formula F, when EPEFV runs to step s, for a clause c and a variable v with the minimum vUnsatT in c, the least upper bound of UnsatT(c, s) is equal to vUnsatT(v, s) in c.*

Proof. In a CNF formula, suppose c includes p+1variables, i.e., c=$v \vee v_1 \vee v_2 \vee \ldots \vee v_p$. For a variable v, there exists at least a clause c where v appears in F. Suppose v appears in t+1 clauses, i.e., c, c_1^v, c_2^v, …, c_t^v. Note that *vUnsatT*(v, s)= *UnsatT*(c, s)+ *UnsatT*(c_1^v, s)+ *UnsatT*(c_2^v, s)+…+ *UnsatT*(c_t^v, s), since *vUnsatT*(v, s)>0 and *vUnsatT* is a nonnegative integer according to its definition in Section 5, and *UnsatT*(c, s)>0 and *UnsatT* is a nonnegative integer based on its definition in Section 4, we can obtain that *vUnsatT*(v, s)≥*UnsatT*(c, s).

Suppose v_1 appears in (t_1) +1 clauses, i.e., c, $c_1^{v_1}$, $c_2^{v_1}$, …, $c_{t_1}^{v_1}$. Note that *vUnsatT*(v_1, s)= *UnsatT*(c, s)+ *UnsatT*($c_1^{v_1}$, s)+ *UnsatT*($c_2^{v_1}$, s)…+ *UnsatT*($c_{t_1}^{v_1}$, s), since *vUnsatT*(v_1, s)>0 and *vUnsatT* is a nonnegative integer, and *UnsatT*(c, s)>0 and *UnsatT* is a nonnegative integer, we can easily obtain *vUnsatT*(v_1, s)≥*UnsatT*(c, s). For the same reason, we can easily obtain *vUnsatT*(v_2, s)≥*UnsatT*(c, s), *vUnsatT*(v_3, s)≥*UnsatT*(c, s),…, *vUnsatT*(v_p, s)≥*UnsatT*(c, s).

As *vUnsatT*(v, s) is the minimum *vUnsatT* in c, min{*vUnsatT*(v, s), *vUnsatT*(v_1, s), *vUnsatT*(v_2, s),…, *vUnsatT*(v_p, s)}= *vUnsatT*(v, s)≥*UnsatT*(c, s). Thus the least upper bound of *c's UnsatT* is *vUnsatT*(v, s) in c. □

We summarize some experimental statistics in Table 5 in order to verify these theoretical expectations (the details of all HRS Testing benchmarks can be seen in Section 7.1), where minΔ_{min} is the minimum values of *vUnsatT* minus *UnsatT*, and maxΔ_{max} is the maximum values of *vUnsatT* minus *UnsatT*. As can be clearly seen from Table 5, minΔ_{min} is equal to or greater than 0 in each class. Thus, the experimental results are consistent with the theoretical ones.

Table 5
The size of *vUnsatT* minus *UnsatT* in an HRS formula. It shows when EPEFV performs step 10000, the experimental minimum and maximum size of *vUnsatT* minus *UnsatT* of each clause on 100 instances for each class.

Variable size	4.3HRS Testing		5.206HRS Testing		5.5HRS Testing		5.699HRS Testing	
	minΔ_{min}	maxΔ_{max}	minΔ_{min}	maxΔ_{max}	minΔ_{min}	maxΔ_{max}	minΔ_{min}	maxΔ_{max}
n=200	0	272	0	813	0	1029	0	920
n=300	0	455	0	509	0	917	0	650
n=400	0	390	0	566	0	648	0	457
n=500	0	364	0	411	0	673	0	559
n=600	0	261	0	299	0	511	0	362
n=700	0	272	0	288	0	362	0	280
n=800	0	259	0	231	0	361	0	281
n=900	0	195	0	221	0	378	0	285
n=1000	0	208	0	221	0	344	0	205

As explained in Section 4, two mechanisms are underlying the EFV heuristic, i.e., the clause selection mechanism including two components: the new clause weighting scheme focusing on the unsatisfied clauses of containing flipping variable and the biased random walk, and the variable selection mechanism only containing one component: the new tie-breaking strategy. Thus, to demonstrate the effectiveness of components (i.e., *vUnsatT* and *UnsatT*) in the EFV heuristic, we conduct experiments to compare EPEFV with the four alternative versions in the following:

- EPEFV_*alt1:* This alternative version of EPEFV does not utilize the new clause weighting scheme, i.e., does not use the *UnsatT* (i.e., removing update clause weights UT of line 14 in Algorithm 2, or replacing the biased random walk component, i.e., lines 6-8 in Algorithm 2, with the standard random walk component, i.e., line 6 in Algorithm 1);
- EPEFV_*alt2:* This alternative version of EPEFV does not use the tie-breaking strategy in the variable selection mechanism. In other words, this alternative version does not select a variable to be flipped based on the U_v function during the search process, i.e., does not use the *vUnsatT*

property (i.e., removing lines 10-12 and update variable property *vUnsatT* of line 14 in Algorithm 2);

- EPEFV_*alt*3*:* This alternative version of EPEFV uses the new tie-breaking strategy in the variable selection mechanism, but the U_v function only uses *score* (i.e., replacing the U_v function, i.e., line 11 and variable property *vUnsatT* of line 14 in Algorithm 2, with the *score*);
- EPEFV_*alt*4 (*ProbSAT*)*:* This alternative version of EPEFV does not use UT and the tie-breaking strategy. i.e., does not use the *UnsatT* property and the *vUnsatT* property (i.e., removing lines 6-8, 10-12, and 14 in Algorithm 2, i.e., replacing Algorithm 2 with Algorithm 1).

To make the experiment more convincing, parameter γ of EPEFV__*alt*1, parameter β of EPEFV__*alt*2 and EPEFV__*alt*3 are set to be the same as those in EPEFV, and we use these parameters in the experiments. Each solver is performed on each instance, with a cutoff time of 600 seconds.

Table 6

Experimental results of EPEFV and its three alternative versions on all testing HRS benchmarks.

Benchmarks	#inst	EPEFV		EPEFV_*alt*1		EPEFV_*alt*2		EPEFV_*alt*3		EPEFV_*alt*4	
		suc	par2	suc	par2	suc	par2	suc	par2	suc	par2
4.3HRS Comp17	40	40	0.023	40	0.006	40	0.009	40	**0.049**	40	0.057
4.3HRS Testing	1000	1000	0.223	960	48.09	1000	**0.021**	1000	0.027	1000	0.023
5.206HRS Comp17	40	40	**0.039**	0	-	39	30.05	40	0.042	0	-
5.206HRS Testing	1000	1000	**0.079**	0	-	1000	0.090	1000	0.089	0	-
5.5HRS Comp17	40	**40**	**0.662**	9	930	22	540.36	29	330.5	9	960
5.5HRS Testing	1000	**1000**	**1.298**	80	1104	360	768.3	280	864.3	160	1008
SAT Comp18	165	**165**	**0.128**	71	683.6	145	1200	152	94.66	79	5259
5.699HRS Testing	900	900	**0.768**	0	-	900	1.074	900	0.973	0	-

Table 7

Implementation rate of three components (*β* of UT, tie-breaking, *r* of tie-breaking) of EPEFV and its three alternative versions on HRS benchmarks.

Benchmarks	EPEFV			EPEFV_*alt*1		EPEFV_*alt2*	EPEFV_*alt*3	
	rate(*β*)	*rate*(*tie*)	*rate*(*r*)	*rate*(*tie*)	*rate*(*r*)	*rate*(*β*)	*rate*(*β*)	*rate*(*tie*)
4.3HRS Comp17	1.000	0.038	0.023	0.037	0.023	0.584	0.656	0.092
4.3HRS Testing	0.998	0.027	0.023	0.033	0.016	0.957	0.983	0.036
5.206HRS Comp17	0.286	0.100	0.087	/	/	0.584	0.284	0.101
5.206HRS Testing	0.291	0.099	0.037	/	/	0.524	0.298	0.100
5.5HRS Comp17	0.528	0.151	0.149	0.053	0.008	0.675	0.598	0.141
5.5HRS Testing	0.540	0.153	0.150	0.062	0.047	0.677	0.547	0.152
5.699HRS Testing	0.371	0.124	0.121	/	/	0.566	0.383	0.126

Empirical results for EPEFV and its four alternative versions on all testing HRS benchmarks (in Section 7.1) are reported in Table 6 and Table 7. Table 6 presents experimental results in term of successful runs and par 2, and Table 7 reports average rate, i.e., steps of performing each component (i.e., steps of *UnsatT* >*β*, steps of performing tie-breaking, the steps of *vUnsatT* >*γ*) divided by total steps for solving each HRS benchmark respectively, denoted by "rate(*β*)", "rate(tie)", and "rate(*γ*)" respectively. If a solver has no successful run for a group of instances, the corresponding average rate' is marked with "/".

By comparing EPEFV and EPEFV_*alt2* in Table 6, EPEFV overall outperforms EPEFV_*alt2* in term of successful runs on each benchmark, and rate(*tie*) of EPEFV is at least 0.1 on HRS instances with $r \approx 5.206$, $r=5.5$ and $r=5.699$ in Table 7, which indicate that the new tie-breaking strategy contributes to the performance of EPEFV on several HRS instances with $r \approx 5.206$, $r=5.5$ and $r=5.699$.

The comparative results of EPEFV_*alt2* and EPEFV_*alt4* shows that EPEFV_*alt2* performs better than EPEFV_*alt4* on all testing HRS benchmarks, and according to Table 7, the rate(β) of EPEFV_*alt2* is larger than 0.50 for each benchmark, which indicates that *UnsatT* property improves ProbSAT on several HRS instances with r=4.3, r≈5.206, r=5.5 and r=5.699. Then the comparison between EPEFV and EPEFV_*alt*3 in Table 6 illustrates that EPEFV performs better than EPEFV_*alt*3 on all testing HRS benchmarks except for the 4.3HRS Testing benchmark. However, the rate(*tie*) of EPEFV_*alt*3 is similar to the rate(*tie*) of EPEFV for each benchmark. Thus, we confirm that *vUnsatT* property plays a significantly important role in EPEFV.

As can be seen from Table 6, the comparison between EPEFV and EPEFV_*alt*1, EPEFV_*alt*2 and EPEFV_*alt*3 shows that EPEFV outperforms its four alternative versions in terms of par 2 on HRS instances with r=5.206, r=5.5 and r=5.699, where the rate(γ) is at least 0.03, while it has similar performance as EPEFV_*alt*2 on HRS instances with r=4.3, where the rate(γ) is smaller than 0.03. We conjecture that the *vUnsatT* property becomes ineffective when the rate(γ) is smaller than 0.03. Furthermore, we will theoretically analyze the influence of *vUnsatT* property on the new tie-breaking strategy.

For a clause c, suppose c={v_1, v_2, v_3}, EPEFV runs to step s, min{*vUnsatT*(v_1, s), *vUnsatT*(v_2, s), *vUnsatT*(v_3, s)}= *vUnsatT*(v_2, s), max{*vUnsatT*(v_1, s), *vUnsatT*(v_2, s), *vUnsatT*(v_3, s)}= *vUnsatT*(v_1, s). If EPEFV needs to perform the new tie-breaking strategy, then EPEFV will select a variable with the maximum U_v value. The difference between U_v (v_1, s) and U_v (v_2, s) is denoted by $\Delta_{12}U$.

$\Delta_{12}U$= U_v (v_1, s)- U_v (v_2, s).

Based on Theorem 1, we can easily obtain the difference between *vUnsatT* and *UnsatT*, denoted by Δ, respectively as follows:

Δ_1=*vUnsatT* (v_1, s-1)- *UnsatT* (c, s-1)

Δ_2=*vUnsatT* (v_2, s-1)- *UnsatT* (c, s-1)

Let Δ_{max} be the maximum difference between *vUnsatT* and *UnsatT* in c, i.e.,

Δ_{max}= Δ_1

Then Let Δ_{12}*score* be the difference between *score* (v_1, s) and *score* (v_2, s), and the difference between Δ_1 and Δ_2 is Δ_{12}, thus

$\Delta_{12}U$= U_v (v_1, s)- U_v (v_2, s)

= *score* (v_1, s) + *vUnsatT* (v_1, s)/γ- *score* (v_2, s) + *vUnsatT* (v_2, s)/γ

= *score* (v_1, s)- *score* (v_2, s)+(*vUnsatT* v_1, s)- *vUnsatT* (v_2, s))/γ

=Δ_{12}*score*+(*UnsatT* (c, s-1)+ Δ_1- *UnsatT* (c, s-1)- Δ_2)/ γ

=Δ_{12}*score*+Δ_{12}/γ≤Δ_{12}*score*+Δ_{max}/γ.

We calculate the difference between U_v (v_1, s) and U_v (v_2, s), which is related to Δ_{12}*score* and γ as well as Δ_{12}. When Δ_{12} is equal to or larger than γ, *vUnsatT* property is able to influence the EPEFV algorithm, i.e., when Δ_{max} is smaller than γ, EPEFV depends only on the *score* function of each variable. These maximum Δ_{max} values are listed in Table 5 for each class with EPEFV running to step 10000.

It is shown in Table 5that the maximum Δ_{max} values of HRS instances with r≈5.206, r=5.5, and r=5.699 are larger than both HRS instances with r=4.3 for each variable size. Thus, the larger the *vUnsatT* value is, the more likely the rate(γ) is at least 0.03, the more effective the EPEFV algorithm will be.

Thus, EPEFV generally is better than its all four alternative versions, which indicates the effectiveness of all components of the proposed EFV heuristic.

9.2 Main Differences between EPEFV and ProbSAT

Although the EPEFV algorithm is conceptually related to the ProbSAT algorithm [6], there exist major differences between EPEFV and ProbSAT. In this subsection, we summarize these major differences as follows:

- **Clause weighting scheme**: EPEFV employs a new clause weighting scheme that only works on the unsatisfied clauses containing the flipping variable, while ProbSAT does not use any clause weighting schemes.
- **Clause selection component**: To give a higher priority to HSC-UT, the EPEFV algorithm applies a biased random walk strategy to select an HSC-UT, while the ProbSAT algorithm uses a standard random walk strategy.
- **Variable selection mechanism**: If the current variable selected based on probability is the same as the last flipped variable, the EPEFV algorithm prefers to select a variable to be flipped by the new tie-breaking strategy preferring the variable with the greatest U_v, while the ProbSAT algorithm does not distinguish the current variable selected and the last flipped variable, if it simply uses the probability function f to pick the variable to be flipped.
- **Empirical performance on HRS benchmarks**: Overall, as can be seen clearly from the extensive experiments illustrated in Tables 2- 5 as well as Table 6, the EPEFV algorithm generally performs much better than the ProbSAT algorithm on a wide range of HRS benchmarks, indicating that the significant performance improvements of EPEFV over ProbSAT are due to the above major differences between these two SLS algorithms.

9.3 Main differences between UT and SWT, PAWS

As the popular clause weighting schemes SWT [14] and PAWS [49] have been successfully applied to SLS algorithms for solving SAT problems, in this subsection, we discuss the main differences between UT and SWT, PAWS for solving HRS instances. Before getting into the details of discussion, we first introduce two clause weighting schemes:

- **PAWS weighting scheme**. PAWS has been used prominently in SLS algorithms for ***picking a variable*** to be flipped [16, 35]. The weight of each clause is a positive integer and is initiated as 1. When a local optimum is reached, the clause weights are updated as follows. With probability p, for each satisfied clause whose weight is large than one, its weight is decreased by one; with probability $(1-p)$, the weights of all unsatisfied clauses are increased by one.
- **SWT weighting scheme**. SWT has been used successfully in SLS algorithms for ***picking a variable*** to be flipped [14]. SWT resembles in some respect the SAPS scheme [26]. The clause weights are updated for solving SAT by SWT as follow: all clause weights are initialized as 1; whenever SWT is called, the weights of all unsatisfied clauses are increased by one; further, if the average weight $\overline{w}$ exceeds a threshold γ, it smooths all clause weights as $w(c)=p\cdot w(c)+ (1-p)\cdot\overline{w}$, where $0<p<1$.

On the one hand, although conventional PAWS and SWT strategies are utilized to pick a variable (such as CSCCSat, Score$_2$SAT, Sparrow, and so on), they become ineffective for solving HRS instances. In order to demonstrate the effectiveness of components in the clause weighting scheme UT, we conduct experiments to compare EPEFV with the two alternative versions in the following.

- EPEFV*_alt5:* This alternative version of EPEFV utilizes PAWS rather than the new clause weighting scheme (i.e., replacing the UT with the PAWS, i.e., line 14 in Algorithm 2, with the PAWS).

- EPEFV_*alt6:* This alternative version of EPEFV utilizes SWT instead of the new clause weighting scheme (i.e., replacing the UT with the SWT, i.e., line 14 in Algorithm 2, with the SWT).

The parameter settings are determined according to our experience for EPEFV's two alternative versions. The parameter settings found for EPEFV_alt5 and EPEFV_alt6 are summarized in Table 8, and we use these parameter settings in the subsequent empirical study.

Table 8

The parameter settings for EPEFV_*alt5* and EPEFV_*alt6* on HRS benchmarks with r=4.3, r=5.206, r=5.5 and r=5.699.

Instance types	variable sizes	EPEFV_*alt5*			EPEFV_*alt6*		
		γ	β	p	γ	β	p
4.3HRS	n<600	1100	100	0.2	1000	200	0.9
	n≥600	200	200	0.9	100	700	0.7
5.206HRS	n<600	800	220	0.9	320	400	0.7
	n≥600	800	80	0.6	800	80	0.6
5.5HRS/5.699HRS	n<600	1500	30	0.1	900	1000	0.4
	n≥600	800	2000	0.1	200	1200	0.2

Table 9

Experimental results of EPEFV and its two alternative versions on all testing HRS benchmark.

Benchmarks	numbers	EPEFV		EPEFV_*alt5*		EPEFV_*alt6*	
		suc	par2	suc	par2	suc	par2
4.3HRS Comp17	40	**40**	0.023	40	0.012	40	0.028
4.3HRS Testing	1000	**1000**	**0.223**	960	48.05	1000	0.406
5.206HRS Comp17	40	**40**	**0.039**	0	-	0	-
5.206HRS Testing	1000	**1000**	**0.079**	0	-	0	-
5.5HRS Comp17	40	**40**	**0.662**	9	987.6	9	987.6
5.5HRS Testing	1000	**1000**	**1.298**	80	1104	80	1104
SAT Comp18	165	**165**	**0.125**	76	649.5	76	647.7
5.699HRS Testing	900	**900**	**0.768**	0	-	0	-

Empirical results for EPEFV and its two alternative versions on all testing HRS benchmarks (in Section 7.1) are reported in Table 9. As can be seen from Table 9, due to the replacement of the SWT or PAWS by the UT, EPEFV_*alt5* and EPEFV_*alt6* perform much worse than the EPEFV on the testing HRS instances with r=4.3, r=5.206, r=5.5, and r=5.699, although EPEFV_*alt5* is faster than EPEFV on solving HRS instances with r=4.3 and n<600, which indicates the effectiveness of the new clause weighting scheme UT and confirms that the proposed EFV heuristic contributes the performance of EPEFV on such instances.

We discuss the main differences between UT and SWT, PAWS in detail as follows.

- **Initial clause weighting scheme**: The initial clause weights of UT are set to be 1 for just the unsatisfied clauses under the initial assignment, while that of SWT and PAWS are set to be 1 for all clauses.
- **Object to update clause weighting scheme**: The clause weighting scheme UT only works for the unsatisfied clauses *containing the current flipping variable*, while SWT works for all unsatisfied clauses and PAWS works for all unsatisfied clauses or satisfied clauses.
- **Goal to activate the clause weighting scheme:** The clause weighting scheme UT is activated to pick a clause, while conventional SWT and PAWS are activated to select a variable (noting that SWT and PAWS are activated to select a clause in EPEFV_*alt5* and EPEFV_*alt6* respectively).
- **Empirical performance on HRS benchmarks**: According to the experimental results presented in Table 9, EPEFV generally outperforms EPEFV_*alt5* and EPEFV_*alt6* in terms of metrics, and thus SWT and PAWS are likely unsuitable for solving HRS instances, indicating that UT gains a

significant improvement over SWT and PAWS for HRS instances with r=4.3, $r \approx 5.206$, r=5.5, and r=5.699.

10. Conclusions and Future Work

In this work, a new selection heuristic with emphasis on the flipping variable, called EFV, was introduced to improve SLS algorithms for solving the HRS as well as URS with long clauses. By incorporating the EFV heuristic into one of the SLS solver ProbSAT, a new SLS algorithm named EPEFV (**E**xtended **P**robability strategy with **E**mphasis on the **F**lipping **V**ariable) was proposed to solve HRS and URS with long clauses.

To demonstrate the effectiveness and the robustness of the EPEFV algorithm, we first evaluated EPEFV on the HRS benchmarks including the latest HRS benchmarks from the random track of SAT Competitions in 2017 and 2018, and additional a broad range of randomly generated HRS benchmarks. Our experimental results showed that EPEFV significantly outperformed the SLS algorithms namely Dimetheus, ProbSAT, YalSAT, Score$_2$SAT and the complete algorithms including Cadical, MapleLCMDist- ChronoBT, gluhack, and SparrowToRiss on all those HRS benchmarks, indicating that EPEFV established a new state-of-the-art on SLS and complete algorithms for solving HRS in terms of success rate and efficiency. Particularly, on several HRS instances with r=4.3, $r \approx 5.206$, r =5.5, and r =5.699, the results showed that the current SLS as well as the complete algorithms either have limited success rates or could not find solutions quickly (e.g., within the cutoff time of 600 CPU seconds), but EPEFV algorithm is able to solve all these HRS benchmarks with 100% success rate and efficient within a few seconds.

Secondly, we conducted experiments on URS benchmarks to compare the EPEFV with state-of-the-art SLS competitors. The experimental results showed that EPEFV significantly outperformed these SLS competitors on solving URS instances with long clauses from the random track of SAT Competitions in 2016 and 2017.

Furthermore, we perform more empirical evaluations to analyze the effectiveness of the EFV heuristic, where the experimental results confirmed the effectiveness of each underlying component in the EFV heuristic and demonstrated that the EFV heuristic contributes to the performance of EPEFV on HRS benchmarks. Also, we conducted empirical evaluations to compare the influence of different clause weighting schemes on the EPEFV, and the experimental results showed that UT is the most suitable clause weighting scheme on the EPEFV algorithm for solving the HRS problems.

Based on the above evaluations, it was clearly evidenced and concluded that the EPEFV can be considered as a new state-of-the-art SLS solver for HRS instances and URS instances with long clauses (The most advanced solvers can only effectively solve URS instances, and the most advanced complete solvers can only effectively solve HRS instances). In further work, we would like to study the flexibility of the proposed EFV heuristic and apply the EFV heuristic to other SLS algorithms and carry out deeper work along this direction. We would also like to combine the EPEFV algorithm and other solvers to achieve better performance on solving HRS instances. A significant research issue is to improve SLS algorithms for structured instances and constrained satisfaction as well as graph search problems by the proposed new heuristics.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (Grant No. 61673320), Sichuan Science and Technology Program (Grant No. 2020YJ0270), Humanities and Social Sciences

Research Project of the Ministry of Education of China (Grant No. 20XJCZH016), and the Fundamental Research Funds for the Central Universities (Grant No. 2682019ZT16, Grant No. 2682020CX5). The authors would like to thank Tomas Balyo for providing the HRS generator.

References

[1] Achlioptas, D. (2009). Random satisfiability. In Handbook of Satisfiability, pp. 245–270.

[2] Balint, A. (2014). Engineering stochastic local search for the satisfiability problem (Doctoral dissertation, Universität Ulm).

[3] Gu, J., Purdom, P. W., Franco, J., & Wah, B. W. (1997). Algorithms for the satisfiability (SAT) problem: A survey. In Satisfiability (SAT) Problem, DIMACS, American Mathematical Society, 1997, pp, 19-151.

[4] Balint, A. and Fröhlich, A. (2010). Improving stochastic local search for sat with a new probability distribution. In Pro. of SAT-2010, pp. 10–15.

[5] Balint, A. and Schöning, U. (2012). Choosing probability distributions for stochastic local search and the role of make versus break. In Pro. of SAT-2012, pp. 16–29.

[6] Balint, A. and Schöning, U. (2018). ProbSAT. In Pro. of SAT-2018: Solver and Benchmark Descriptions, pp. 35.

[7] Balint, A. and Manthey, N. (2018). SparrowToRiss. In Proc. of SAT 2018: Solver and Benchmark Descriptions, pp, 38-39.

[8] AlKasem, H. H., & Menai, M. E. B. (2020). Stochastic local search for Partial Max-SAT: an experimental evaluation. Artificial Intelligence Review, pp. 1-42.

[9] Balyo, T, Chrpa L. (2018). Using Algorithm Configuration Tools to Generate Hard SAT Benchmarks. In Proceedings of Eleventh Annual Symposium on Combinatorial Search, pp. 133-137.

[10] Barthel, W., Hartmann, A. K., *et al* (2002). Hiding solutions in random satisfiability problems: A statistical mechanics approach. Physical review letters, vol. 88, no. 18, pp. 188701.

[11] Biere A. (2017). Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2017. In Pro. of SAT-2017: Solver and Benchmark Descriptions, pp.14-15.

[12] Bearden S R B , Pei Y R , Ventra M D (2020). Efficient solution of Boolean satisfiability problems with digital memcomputing. Scientific Reports, vol.10, no.1.

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

[14] Cai, S., & Su, K. (2013). Local search for Boolean Satisfiability with configuration checking and subscore. Artif. Intell. 204, 75-98.

[15] Cai, S., & Su, K. (2013). CCAnr. In Pro. of SAT-2013: Solver and Benchmark Descriptions, pp. 16–17.

[16] Cai, S., Luo, C., & Su, K. (2014). Scoring functions based on second level score for k-SAT with long clauses. Journal of Artificial Intelligence Research, vol. 51, pp. 413-441.

[17] Cai, S. & Luo, C. (2017). Score$_2$SAT: Solver description. In Pro. of SAT-2017: Solver and Benchmark Descriptions, pp. 34.

[18] Alouneh, S., Al Shayeji, M. H., & Mesleh, R. (2019). A comprehensive study and analysis on SAT-solvers: advances, usages and achievements. Artificial Intelligence Review, vol.52, no. 4, pp. 2575-2601.

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

[20] Duong, T. T. N., Pham, D. N., Sattar, A., & Newton, M. H. (2013). Weight-enhanced diversification in stochastic local search for satisfiability. In Proc. of IJCAI 2013, pp.524-530.

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

[22] Gableske, O. (2018). Dimetheus. In Pro. of SAT-2018: Solver and Benchmark Descriptions, pp. 20-21.

[23] Hoos, HH. (2002). An adaptive noise mechanism for WalkSAT. In Pro. of AAAI'02, pp. 655–660.

[24] Fu, H., Wu, G., Liu, J., & Xu, Y. (2020). More efficient stochastic local search for satisfiability. Applied Intelligence, pp.1-20.

[25] Barthel, W., Hartmann, A. K., Leone, M., et al (2002). Hiding solutions in random satisfiability problems: A statistical mechanics approach. Physical review letters, vol. 88, no. 18, pp. 188701, 2002.

[26] Hutter, F., Tompkins, D. A., & Hoos, H. H. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In Proc. of CP 2002, pp, 233-248.

[27] Kirkpatrick, S., & Selman, B. (1994). Critical behavior in the satisfiability of random Boolean formulae. Science, 264, 1297–1301

[28] Biere A. (2019). CADICAL at the SAT Race 2019. In Pro. of SAT-2019: Solver and Benchmark Descriptions, pp. 8-9.

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

[30] Kroc, L., Sabharwal, A., & Selman, B. (2010). An empirical study of optimal noise and runtime distributions in local search. In Proc. of SAT-10, pp. 346–351.

[31] Li, C. & Li Y. (2012). Satisfying versus falsifying in local search for satisfiability - (poster presentation). In Proc. of SAT-2012, pp. 477–478.

[32] Li, C. & Huang, W. (2005). Diversification and determinism in local search for satisfiability. In Pro. of SAT'05, pp. 158–172.

[33] Liang, J., Ganesh, V., Poupart, P., Czarnecki, K. (2016). Learning rate based branching heuristic for sat solvers. In Proc. of SAT-2016, pp. 123-140.

[34] Liang, J., Ganesh, V., Poupart, P., Czarnecki, K. (2017). An empirical study of branching heuristics through the lens of global learning rate. In Proc. of SAT-2017, pp. 119-135.

[35] Luo, C., Cai, S., Wu, W., & Su, K. (2014). Double configuration checking in stochastic local search for satisfiability. In Pro. of AAAI-2014, pp. 2703-2709.

[36] Luo, C., Cai, S., Wu, W., and Su, K. (2016). CSCCSat2014. In Pro. of SAT-2016: Solver and Benchmark Descriptions, pp. 10.

[37] Luo, C., Cai, S. and Su, K.. (2016). DCCAlm. In Pro. of SAT-2016: Solver and Benchmark Descriptions, pp. 11.

[38] Cai, S., Y., Hou, W. and Wang, H. (2019). Towards faster local search for minimum weight vertex cover on massive graphs. Inf. Sci., 471, 64–79.

[39] Luo, C., Cai, S., Su, K.. & Huang, W. (2017). CCEHC: An Efficient Local Search Algorithm for Weighted Partial Maximum Satisfiability (Extended Abstract). In Proc. of IJCAI, pp, 5030-5034.

[40] Kyrillidis, A., Shrivastava, A., Vardi, M., & Zhang, Z. (2020, April). FourierSAT: A Fourier Expansion-Based Algebraic Framework for Solving Hybrid Boolean Constraints. In Proc. of IJCAI 2020, pp. 1552-1560.

[41] Manthey, N. (2012). Coprocessor 2.0: a flexible cnf simplifier. In Proc. of SAT-2012, pp. 436–441.

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

[43] Mitchell, D., Selman, B., & Levesque, H. (1992). Hard and easy distributions of SAT problems. In Proc. of AAAI, Vol. 92, pp. 459-465.

[44] Zha, A. (2018). GluHack. In Proc. of SAT-2018: Solver and Benchmark Descriptions, pp. 26.

[45] Ouimet, M., & Lundqvist, K. (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.

[46] Ryvchin, V. & Nadel, A. (2018). MapleLCMDistChronoBT. In Proc. of SAT-2018: Solver and Benchmark Descriptions, pp. 29.

[47] Selman, B., Kautz, H. A., & Cohen, B. (1994). Noise strategies for improving local search. In Proc. of AAAI 1994, pp, 337–343.

[48] Selman, B. (1995). Stochastic search and phase transitions: AI meets physics. In Proc. of IJCAI (1), pp. 998-1002.

[49] Thornton, J. (2005). Clause weighting local search for SAT. Journal of Automated Reasoning, 35(1-3), 97-142.

[50] Wu, Z., & Wah, B. W. An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In Proc. of AAAI/IAAI, 2000, pp, 310-315.

Appendix

This appendix section provides all the tables involved in the experimental section in the paper.

Table 1

Computational results on the HRS instances with r=4.3.

Benchmark	Instance Class		Score$_2$SAT	YalSAT	Dimetheus	ProbSAT	gluHack	STR	MBT	Cadical	EPEFV
4.3HRS Comp17	n=400	suc	50	50	50	50	50	50	50	50	50
		par2	**0.003**	0.08	0.021	0.002	261.2	0.090	102.6	274.6	0.004
	n=420	suc	50	50	50	50	50	50	50	50	50
		par2	0.012	0.014	0.033	0.002	481.1	0.080	390.6	481.7	0.010
	n=440	suc	50	50	50	50	30	50	40	40	50
		par2	**0.020**	0.072	0.326	0.448	4778	0.351	2464	2616	0.130
	n=460	suc	50	50	50	50	50	50	30	50	50
		par2	0.012	0.020	0.017	**0.006**	1572	0.078	4241	278.7	0.012
	n=480	suc	50	50	50	50	40	50	30	50	50
		par2	**0.006**	0.204	0.017	0.012	2481	0.079	4780	181.3	0.012
	n=500	suc	50	50	50	50	20	50	30	50	50
		par2	0.003	0.002	0.016	0.009	6481	0.068	4314	575.8	0.002
	n=520	suc	50	50	50	50	30	50	30	40	50
		par2	0.010	0.010	0.014	0.009	4939	0.088	4765	2360	0.002
	n=540	suc	50	50	50	50	20	50	10	20	50
		par2	0.014	0.006	0.013	**0.006**	6908	0.101	8034	6225	0.010
	Overall	suc	400	400	400	400	290	400	270	350	400
		par2	0.020	**0.017**	0.057	0.062	3481	0.117	3641	1625	0.023
4.3HRS Testing	n=600	suc	200	200	200	200	40	200	0	80	200
		par2	0.007	0.004	0.009	0.007	976.5	0.051	-	794.0	0.014
	n=700	suc	200	200	200	200	0	200	0	0	200
		par2	0.019	0.023	0.033	**0.010**	-	0.165	-	-	0.028
	n=800	suc	200	200	200	200	0	200	0	0	200
		par2	0.052	**0.019**	0.055	0.030	-	0.373	-	-	0.771
	n=900	suc	200	200	200	200	0	200	0	0	200
		par2	0.044	0.028	**0.022**	0.072	-	1.120	-	-	0.281
	n=1000	suc	200	200	200	200	0	200	0	0	200
		par2	0.016	0.010	0.022	**0.002**	-	0.172	-	-	0.022
	Overall	suc	1000	1000	1000	1000	40	1000	0	80	1000
		par2	0.028	**0.017**	0.028	0.024	1347	0.376	-	1119	0.223

Experiments on the HRS instances with r=4.3

Table 1 in the Appendix presents the comparative results of EPEFV with its SLS competitors Score$_2$SAT, YalSAT, Dimetheus as well as ProbSAT and complete competitors gluHack, STR, MBT and Cadical on 4.3HRS Comp17 benchmark and 4.3HRS testing benchmark. According to **Table 1 in**

the Appendix, EPEFV outperforms all complete competitors gluHack, MBT and Cadical in terms of metrics. Although EPEFV is slower than Score$_2$SAT, Dimetheus, YalSAT and ProbSAT in terms of par 2, EPEFV, Score$_2$SAT, YalSAT, Dimetheus and ProbSAT show the same performance in terms of successful runs. Overall, EPEFV outperforms STR in terms of par 2.

Table 2

Computational results on the HRS instances with r=5.206.

Benchmark	Instance Class	Score$_2$SAT	YalSAT	Dimetheus	ProbSAT	gluHack	STR	MBT	Cadical	EPEFV
		suc	suc	suc	suc	suc	suc	suc	suc	suc
		par2	par2	par2	par2	par2	par2	par2	par2	par2
5.206HRS Comp17	n=400	0	0	0	0	50	50	50	50	50
		-	-	-	-	45.87	2.052	4.698	15.96	**0.034**
	n=420	0	0	0	0	50	50	50	50	50
		-	-	-	-	19.72	0.976	16.27	114.0	**0.032**
	n=440	20	0	0	0	50	50	50	50	50
		6000	-	-	-	44.83	0.659	35.70	147.4	**0.034**
	n=460	0	0	0	0	50	50	50	50	50
		-	-	-	-	259.2	0.964	32.61	128.2	**0.042**
	n=480	10	0	0	0	50	50	50	50	50
		8000	-	-	-	526.1	13.75	23.55	703.9	**0.038**
	n=500	0	0	0	0	50	50	50	50	50
		-	-	-	-	278.2	1.505	44.37	915.0	**0.038**
	n=520	0	0	0	0	40	50	50	50	50
		-	-	-	-	2656	17.17	38.44	1587	**0.050**
	n=540	0	0	0	0	40	50	50	30	50
		-	-	-	-	3074	8.603	345.5	5819	**0.044**
	Overall	30	0	0	0	380	400	400	380	400
		9250	-	-	-	863.2	5.709	67.64	1179	**0.039**
5.206HRS Testing	n=600	0	0	0	0	80	200	120	80	200
		-	-	-	-	815.2	11.25	537.8	1870	**0.048**
	n=700	0	0	0	0	0	200	33	40	200
		-	-	-	-	-	11.81	1035	1515	**0.058**
	n=800	0	0	0	0	0	160	40	0	**200**
		-	-	-	-	-	273.0	1075	-	**0.089**
	n=900	0	0	0	0	0	120	0	0	**200**
		-	-	-	-	-	505.2	-	-	**0.091**
	n=1000	0	0	0	0	0	120	0	0	**200**
		-	-	-	-	-	504.5	-	-	**0.110**
	Overall	0	0	0	0	80	800	193	120	**1000**
		-	-	-	-	323.2	261.2	1010	1642	**0.079**

Experiments on the HRS instances with r≈5.206

Table 2 in the Appendix presents the experimental results of EPEFV and its competitors on the 5.206HRS Comp17 benchmark and 5.206HRS Testing benchmark. EPEFV stands out as the best solver in terms of successful runs and par 2 on these benchmarks. Overall, EPEFV solves each instance within one second. It is even more promising that EPEFV is over 146 times faster than STR in overall 5.206HRS Comp17 benchmarks. It is promising to see the performance of EPEFV remains surprisingly good on 5.206HRS testing benchmark, where its competitors show rather poor performance, especially for SLS solvers. For 5.206HRS testing benchmark, on all 1000 instances, EPEFV found the solution for 1000 of them, while the results for Score$_2$SAT, YalSAT, Dimetheus, ProbSAT, gluHack, Cadical, MBT and STR are only 0, 0, 0, 0, 80, 120, 193 and 800 respectively. Furthermore, EPEFV succeeded in 200 runs for the HRS instances with n=900 and n=1000, although the competitor STR solves 120 runs on these two classes of instances, whereas all other competitors failed to find a solution for any of these instances.

STR won the random track of SAT Competition 2018 and gluHack also exhibits good performance on this benchmark, so it is challenging to improve such performance over STR and gluHack on the HRS instances with r=5.206. Indeed, to the best of our knowledge, all 5.206HRS testing instances are solved for the first time. Given the good performance of EPEFV on the 5.206HRS Testing benchmark with 1000 variables, it is very likely to be able to solve larger HRS instances with r=5.206. The experimental results show that EPEFV algorithm achieves the advanced performance on HRS instances with r=5.206.

Table 3

Computational results on the HRS instances with r=5.5.

Benchmark	Instance Class		Score$_2$SAT	YalSAT	Dimetheus	ProbSAT	gluHack	STR	MBT	Cadical	EPEFV
5.5HRS Comp17	n=400	suc	10	10	10	10	50	50	50	50	50
		par2	800	800	800	800	8.608	154.3	2.153	4.938	**0.468**
	n=420	suc	20	20	20	20	50	50	50	50	50
		par2	6000	6000	6000	6000	3.801	102.6	2.217	3.424	**0.548**
	n=440	suc	0	0	0	0	50	50	50	50	50
		par2	-	-	-	-	6.723	194.1	5.583	4.090	**0.628**
	n=460	suc	10	10	10	10	50	50	50	50	50
		par2	8000	8000	8000	8000	33.22	157.4	3.616	23.80	**0.746**
	n=480	suc	10	10	10	10	50	50	50	50	50
		par2	8000	8000	8000	8000	29.7	149.0	3.599	8.626	**0.570**
	n=500	suc	10	20	20	20	50	50	50	50	50
		par2	8000	6000	6000	6000	45.72	122.4	3.370	17.05	**0.460**
	n=520	suc	20	10	10	10	50	50	50	50	50
		par2	6000	8000	8000	8000	33.91	158.5	5.128	19.82	**0.850**
	n=540	suc	10	10	10	10	50	50	50	50	50
		par2	8000	8000	8000	8000	37.54	169.8	4.610	40.08	**1.022**
	Overall	suc	90	90	90	90	400	400	400	400	400
		par2	7750	7750	7750	7750	24.90	151.0	3.785	15.23	**0.662**
5.5HRS Testing	n=600	suc	40	40	40	40	160	200	400	200	200
		par2	960.0	960.0	960.0	960.0	296.0	5.812	4.002	136.0	**0.781**
	n=700	suc	40	40	40	40	120	140	200	200	200
		par2	960.0	960.0	960.0	960.0	581.4	363.6	62.33	157.3	**0.950**
	n=800	suc	40	40	40	40	0	40	160	40	**200**
		par2	960.0	960.0	960.0	960.0	-	960.4	254.7	1065	**1.396**
	n=900	suc	40	40	40	40	40	80	160	0	**200**
		par2	960.0	960.0	960.0	960.0	1050	722.2	350.0	-	**1.594**
	n=1000	suc	0	0	0	0	40	0	200	0	200
		par2	-	-	-	-	1072	-	119.3	-	**1.767**
	Overall	suc	160	160	160	160	360	460	920	440	**1000**
		par2	1008	1008	1008	1008	839.8	650.3	158.1	751.6	**1.298**

Experiments on the HRS instances with r=5.5

Table 3 in the Appendix summarizes the experimental results on the 5.5HRS Comp17 benchmark and 5.5HRS Testing benchmark. It is clear that EPEFV shows significantly better performance than all its competitors on the whole benchmark. EPEFV is the only solver that solves all 5.5HRS Comp17 benchmark and 5.5HRS random benchmark in all runs. Also, EPEFV outperforms its competitors in terms of par 2, which is more obvious as the instance size increases. In particular, on the instances with n=1000, which are of the large size on 5.5HRS Testing benchmark, the runtime of EPEFV is 2 orders of magnitudes less than that of complete solvers, and 3 orders of magnitudes less than that of SLS solvers, which illustrates its robustness. The experimental results show that EPEFV achieves the advanced performance on HRS instances with r=5.5.

Table 4

Computational results on the **SAT Comp18** benchmark.

Instance Ratio	$Score_2SAT$ suc par2	YalSAT suc par2	Dimetheus suc par2	ProbSAT suc par2	gluHack suc par2	STR suc par2	MBT suc par2	Cadical suc par2	EPEFV suc par2
r=4.3	55	55	55	55	55	55	55	55	55
	0.001	**0.001**	0.007	0.013	10.98	0.052	8.636	53.76	0.007
r=5.206	33	27	12	12	55	55	55	55	55
	4000	5228	7858	7985	3.425	1.020	3.281	7.436	**0.019**
r=5.5	12	12	12	12	55	55	55	55	55
	7818	7818	7818	7818	2.035	136.4	1.744	1.067	**0.356**
Overall	100	94	79	79	165	165	165	165	165
	3939	4349	5526	5259	5.476	45.82	4.544	21.13	0.**128**

Experiments on the SAT Comp18 benchmark

To investigate the performance of EPEFV on random HRS benchmarks with various ratio, we compare it with its SLS and complete competitors on all HRS instances with r=4.3, r=5.206 and r=5.5 from SAT Competition 2018. **Table 4 in the Appendix** reports the number of solved instances and par 2 for each solver on each HRS benchmark. Since ProbSAT is the basic framework of EPEFV, we first compare these two solvers. As shown in **Table 4 in the Appendix**, ProbSAT solves 79 HRS instances, while EPEFV solves 165 HRS instances, which is 2 times as that solved by ProbSAT in overall HRS instances.

EPEFV solved more instances than all SLS competitors. Overall, EPEFV solved 165 HRS instances, compared to 79 for Dimetheus, and 94 for YalSAT, and 100 for $Score_2SAT$. Further observation shows that, although EPEFV solves the same number of? instances as the ones solved by all complete competitors and SparrowToRiss, EPEFV is about 36 times faster than MBT (MBT performs the least time among those competitors). In particular, EPEFV has similar performance with $Score_2SAT$, YalSAT and Dimetheus on HRS instances with r=4.3, and significantly outperforms gluHack, STR, MBT and Cadical on HRS instances with r=4.3, r=5.206 and r=5.5.

Table 5

Computational results on the **5.699HRS Testing** benchmark.

Instance Class	$Score_2SAT$ suc par2	YalSAT suc par2	Dimetheus suc par2	ProbSAT suc par2	gluHack suc par2	STR suc par2	MBT suc par2	Cadical suc par2	EPEFV suc par2
n=200	0	0	0	0	100	100	100	100	100
	-	-	-	-	**0.027**	42.01	1.217	0.109	0.202
n=300	0	0	0	0	100	100	100	100	100
	-	-	-	-	0.437	94.28	1.688	1.098	**0.282**
n=400	0	0	0	0	100	100	100	100	100
	-	-	-	-	2.372	213.5	1.994	3.497	**0.471**
n=500	0	0	0	0	100	100	100	100	100
	-	-	-	-	34.44	228.2	2.862	8.975	**0.607**
n=600	0	0	0	0	60	80	100	100	100
	-	-	-	-	509.5	444.1	8.714	104.8	**0.783**
n=700	0	0	0	0	100	40	100	59	100
	-	-	-	-	190.4	855.9	153.6	542.3	**0.910**
n=800	0	0	0	0	40	0	100	0	100
	-	-	-	-	804.8	-	104.97	-	**1.025**
n=900	0	0	0	0	0	0	80	0	**100**
	-	-	-	-	-	-	367.7	-	**1.246**
n=1000	0	0	0	0	0	0	20	0	**100**
	-	-	-	-	-	-	974.2	-	**1.388**
Overall	0	0	0	0	600	520	800	559	**900**
	-	-	-	-	438.1	608.7	179.7	473.4	**0.768**

Experiments on the 5.699HRS testing benchmark

Table 5 in the Appendix reports the experimental results for each solver on 5.699HRS testing benchmark. For the instances with n=200, EPEFV is slower than gluHack, but EPEFV and gluHack solved the same number of instances. For the instances with n=300, n=400, n=500, n=600 and n=700, EPEFV and MBT solved the same number of instances, but EPEFV has less accumulative run time. For the instances with n=900 and n=1000, EPEFV solved the most instances. Especially, EPEFV shows significantly superior performance than its competitors on the instances with n=1000, where it solved 100 instances, while MBT solved 20 instances and other competitors failed to find a solution for any of these instances. Overall, EPEFV solved 900 instances, compared to 0, 0, 0, 0, 600, 520, 559 and 800 instances for Score$_2$SAT, YalSAT, Dimetheus, ProbSAT, gluHack, STR, Cadical and MBT respectively.

Table 6

Computational results on the **SAT Competition 2017** benchmark.

Instance types	Variable sizes and ratios	Dimetheus		ProbSAT		YalSAT		Score$_2$SAT		EPEFV	
		suc	par2	suc	par2	suc	par2	suc	par2	suc	par2
5-SAT	n<600, r=21.117	121	7032	132	6829	130	6880	140	6655	**170**	**6074**
	n=250000, r<21.117	130	3755	110	4526	120	4147	80	6231	130	**3667**
7-SAT	n<200, r=87.79	181	5552	181	5791	170	5957	193	5582	**201**	**5499**
	n=50000, r<87.79	**160**	**2117**	110	4514	90	5517	110	5756	120	4015
Overall		592	5173	533	5713	510	5890	534	6077	**621**	**5138**

Experiments on the SAT Competition 2017 benchmark

Table 6 in the Appendix presents the results of the performance of EPEFV compared with the current SLS solvers on all URS instances with long clauses from SAT Competition 2017. The results show that for huge 7-SAT instances with n=50000 and r<87.79, the performance of EPEFV and Dimetheus are similar and better than that of other competitors, and for the remaining instances class, EPEFV significantly outperforms its competitors in terms of successful runs and par 2.

Especially, EPEFV succeeds in a few more runs than ProbSAT and Score$_2$SAT on random 5-SAT instances at phase transition. EPEFV succeeds in 170 runs, compared to 132 for ProbSAT and 140 for Score$_2$SAT. Further observation shows that EPEFV succeeds in 201 runs, compared to 181 for Dimetheus and ProbSAT and 193 for Score$_2$SAT on random 7-SAT instances at phase transition. Overall, EPEFV succeeds in 621 runs, whereas none of its competitors succeeds in more than 600 runs with the cutoff time, which illustrates its robustness.

Table 7

Computational results on the **5-SAT medium** benchmark.

Instances class	Dimetheus		ProbSAT		YalSAT		Score$_2$SAT		EPEFV	
	suc	suc	suc	par2	suc	par2	suc	par2	suc	par2
5-SAT-200	11	4506	11	4505	11	4513	11	4502	11	**4500**
5-SAT-250	10	**5009**	10	5158	10	5247	9	5517	9	5518
5-SAT-300	10	**5171**	8	6116	10	5283	9	6060	10	5381
Overall	31	**4895**	29	5260	31	5014	29	5360	30	5133

Experiments on the 5-SAT medium benchmark

Table 7 in the Appendix reports the experimental results for each solver on 5-SAT medium benchmark. For medium 5-SAT instances with n=200, EPEFV gives the best performance. Overall, EPEFV significantly outperforms ProbSAT and Score$_2$SAT on this benchmark, and has similar

performance as the best solver Dimetheus, solving only one less instance.

Table 8

Computational results on the **7-SAT medium** benchmark.

Instances class	Dimetheus		ProbSAT		YalSAT		$Score_2SAT$		EPEFV	
	suc	suc	suc	suc	suc	suc	suc	par2	suc	par2
7-SAT-100	12	4057	12	4044	12	4044	12	**4043**	11	4503
7-SAT-110	11	4562	11	**4559**	11	4749	11	4580	9	5518
7-SAT-120	10	5432	9	5969	10	5451	10	**5242**	10	5381
Overall	33	4690	32	4857	33	4857	**33**	**4622**	30	5134

Experiments on the 7-SAT medium benchmark

Table 8 in the Appendix presents the results of the performance of EPEFV compared with the current SLS solvers on the 7-SAT medium benchmark. As can be seen from **Table 8**, EPEFV has similar performance with all SLS competitors on this benchmark.

Table 9

Computational results on the **SAT Competition 2016** benchmark.

Instances class	Dimetheus		ProbSAT		YalSAT		$Score_2SAT$		EPEFV	
	suc	par2	suc	par2	suc	par2	suc	par2	suc	par2
5-SAT r=21.115	131	6891	141	6716	130	6913	132	6797	**150**	**6445**
7-SAT r=87.79	180	5876	180	5762	152	6401	179	6298	180	**5759**
Overall	311	6383	321	6239	282	6657	311	4808	**330**	**6102**

Experiments on the SAT Competition 2016 medium benchmark

Table 9 in the Appendix presents the experimental results of EPEFV and its competitors on URS instances at phase transition from SAT Competition 2016. Since EPEFV is based on ProbSAT, we first compare these two solvers. As can be seen form **Table 9**, EPEFV succeeds in more runs than ProbSAT on all instances calssess. Overall, ProbSAT succeeds in 321 runs, while EPEFV succeeds in 330 runs. EPEFV succeeds in a few more runs than its competitors. Overall, EPEFV succeeds in 330 runs, compared to 311 for both Dimetheus and $Score_2SAT$ and 282 for YalSAT.

Table 10

Computational results on the huge k-SAT instances with k=5, 7.

k-SAT	Instances class	Dimetheus		ProbSAT		YalSAT		$Score_2SAT$		EPEFV	
		suc	par2	suc	par2	suc	par2	suc	par2	suc	par2
5-SAT	5-SAT- r=18.0 *n=250000*	50	**673.2**	48	819.0	50	344.7	50	3780	50	904.2
	5-SAT- r=18.2 n=250000	50	**464.6**	0	-	46	1635	0	-	50	1632
	5-SAT- r=18.4 n=250000	50	**741.8**	0	-	0	-	0	-	50	2120
	overall	150	**626.5**	48	7073	96	4260	50	7927	150	1552
7-SAT	7-SAT- r=64.0 *n=50000*	50	114.7	50	**37.68**	21	6363	50	1890	50	103.7
	7-SAT- r=65.0 *n=50000*	50	392.3	47	797.4	0	-	41	5220	50	**235.5**
	7-SAT- r=66.0 *n=50000*	50	**449.8**	0	-	0	-	0	-	50	1080
	Overall	150	**318.9**	97	3812	21	10721	91	6303	150	473.1

Experiments on the k-SAT huge instances with k=5, 7

The huge sized instances with a few million clauses and the ratio from far from the phase-transition ratio to relatively close, are as large as some of the application benchmarks. We compare EPEFV with SLS solvers on huge 5-SAT and 7-SAT instances.

As can be seen form **Table 10 in the Appendix**, EPEFV is based on ProbSAT, while EPEFV solves more instances than ProbSAT. Overall, ProbSAT solves 48 (out of 150) and 97 (out of 150) instances for huge 5-SAT and 7-SAT instacnes respectively, while EPEFV solves all huge instances, which is 3 times as that solved by ProbSAT on huge 5-SAT instacnes. Dimetheus and EPEFV solve more instances than YalSAT and Score$_2$SAT. EPEFV has similar performance with Dimetheus on this benchmark, solving all instances.