Journal of Artificial Intelligence Research 51 (2014) 413-441 Submitted 06/14; published 10/14

Scoring Functions Based on Second Level Score for k-SAT with Long Clauses

Shaowei Cai SHAOWEICAI.CS@GMAIL.COM
State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing, China
Queensland Research Lab, NICTA, Brisbane, Australia

Chuan Luo CHUANLUOSABER@GMAIL.COM
Key Laboratory of High Confidence Software Technologies,
Peking University, Beijing, China

Kaile Su K.SU@GRIFFITH.EDU.AU
Institute for Integrated and Intelligent Systems,
Griffith University, Brisbane, Australia

Abstract

It is widely acknowledged that stochastic local search (SLS) algorithms can efficiently find models for satisfiable instances of the satisfiability (SAT) problem, especially for random k-SAT instances. However, compared to random 3-SAT instances where SLS algorithms have shown great success, random k-SAT instances with long clauses remain very difficult. Recently, the notion of second level score, denoted as $score_2$, was proposed for improving SLS algorithms on long-clause SAT instances, and was first used in the powerful CCASat solver as a tie breaker.

In this paper, we propose three new scoring functions based on $score_2$. Despite their simplicity, these functions are very effective for solving random k-SAT with long clauses. The first function combines $score$ and $score_2$, and the second one additionally integrates the diversification property age. These two functions are used in developing a new SLS algorithm called CScoreSAT. Experimental results on large random 5-SAT and 7-SAT instances near phase transition show that CScoreSAT significantly outperforms previous SLS solvers. However, CScoreSAT cannot rival its competitors on random k-SAT instances at phase transition. We improve CScoreSAT for such instances by another scoring function which combines $score_2$ with age. The resulting algorithm HScoreSAT exhibits state-of-the-art performance on random k-SAT ($k > 3$) instances at phase transition. We also study the computation of $score_2$, including its implementation and computational complexity.

1. Introduction

The Boolean Satisfiability (SAT) problem is a prototypical NP-complete problem whose task is to decide whether the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. This problem plays a prominent role in various areas of computer science and artificial intelligence, and has been widely studied due to its significant importance in both theory and applications.

Two popular approaches for solving SAT are conflict driven clause learning (CDCL) and stochastic local search (SLS). The latter operates on complete assignments and tries to find a model by iteratively flipping a variable. Although SLS algorithms are typically incomplete in the sense that they cannot prove an instance to be unsatisfiable, they often find models of satisfiable formulas surprisingly effectively.

Most SLS algorithms for SAT switch between two different modes, i.e., the greedy (intensification) mode and the diversification mode. In the greedy mode, they prefer to flip variables whose flips can decrease the number of falsified clauses; in the diversification mode, they tend to better explore the search space and avoid local optima, usually using randomized strategies and exploiting diversification properties of variables to pick a variable for this aim.

SLS is well known as the most effective approach for solving random satisfiable instances, and SLS algorithms are often evaluated on uniform random k-SAT benchmarks. These benchmarks have a large variety of instances to test the robustness of algorithms, and by controlling the instance size and the clause-to-variable ratio, they provide adjustable hardness levels to assess the solving capabilities. Moreover, the performance of algorithms are usually stable on random k-SAT instances, either good or bad. Thus, we can easily recognize good heuristics by testing SLS algorithms on random k-SAT instances, and these heuristics may be beneficial for solving realistic problems. Numerous works have been devoted to designing SLS algorithms for random k-SAT instances with a clause-to-variable ratio at or near the solubility phase transition, which are the most difficult among random k-SAT instances (Kirkpatrick & Selman, 1994).

Among random k-SAT instances, random 3-SAT ones exhibit some particular statistical properties and are easy to solve, for example, by SLS algorithms and a statistical physics approach called Survey Propagation (Braunstein, Mézard, & Zecchina, 2005). It has been shown that the famous SLS algorithm WalkSAT (Selman, Kautz, & Cohen, 1994), which was proposed two decades ago, scales linearly with the number of variables for random 3-SAT instances near the phase transition and can solve such instances with one million variables (Kroc, Sabharwal, & Selman, 2010). The latest state of the art in this direction is an SLS algorithm called FrwCB, which solves random 3-SAT instances near the phase transition (at ratio 4.2) with millions of variables within 2-3 hours (Luo, Cai, Wu, & Su, 2013).

In contrast, random k-SAT instances with long clauses remain very difficult, and the performance of SLS algorithms on such instances has stagnated for a long time. Indeed, such instances are challenging for all kinds of algorithms, including the Survey Propagation algorithm, which solves random 3-SAT instances extremely fast (Mézard, 2003) and is also adapted for solving MaxSAT (Chieu & Lee, 2009). Recently, a few progresses such as Sattime (Li & Li, 2012), probSAT (Balint & Schöning, 2012) and CCASat (Cai & Su, 2013b), have been made in this direction. In particular, when solving random instances near the phase transition, the Sattime algorithm is good at solving random 6-SAT and 7-SAT instances, and probSAT is good at solving random 4-SAT and 5-SAT instances. Comparatively, CCASat shows good performance on all random k-SAT instances for $k \in \{4, 5, 6, 7\}$ and won the random track of SAT Challenge 2012. Note that the second and third solvers in that track are variants of the portfolio solver SATzilla (Xu, Hutter, Hoos, & Leyton-Brown, 2008). On the other hand, probSAT and Sattime show better performance than CCASat on random k-SAT instances at the threshold ratio of phase transition.

A key notion in CCASat is the $score_2$ property[1], which shares the same spirit with the commonly used $score$ property and can be regarded as the second level score. It considers transformations between clauses with one true literal and those with two true literals. By breaking ties using $score_2$, the performance of CCASat is significantly improved for random k-SAT instances

1. the $score_2$ property is denoted by subscore in CCASat (Cai & Su, 2013b).

with $k > 3$ (Cai & Su, 2013b). This leads us to such a question — *Can we further improve SLS algorithms on such instances by making better use of the $score_2$ property?* In this paper, we give a positive answer to this question by proposing three new scoring functions based on $score_2$, and using them to develop two SLS algorithms which outperform state-of-the-art solvers on random k-SAT with $k > 3$ near and at phase transition.

The first scoring function proposed in this paper is called $cscore$, which is a linear combination of $score$ and $score_2$. The $cscore$ function differs from previous hybrid scoring functions in that it considers two "score" properties of different levels. Based on this scoring function, we also define a new type of "decreasing" variables namely comprehensively decreasing variables. The $cscore$ function enhances intensification in the greedy mode by integrating the current greediness and the look-ahead greediness. Further, by combining $cscore$ with the diversification property age (the definition of age can be found in Section 2.1), we propose the second scoring function dubbed $hscore$, which is used to improve the diversification mode. These two scoring functions are used to develop an SLS algorithms called CScoreSAT.

We conduct extensive experiments to compare CScoreSAT against state-of-the-art SLS solvers including winners from the most recent SAT competitions. The experiments on large random 5-SAT and 7-SAT instances near phase transition show that CScoreSAT significantly outperforms its competitors in terms of success rate or run time. In particular, CScoreSAT is able to solve random 5-SAT instances with up to 5000 variables and random 7-SAT instances with up to 300 variables, whereas all its competitors fail to solve such instances of this size.

However, the performance of CScoreSAT on random k-SAT instances at the threshold ratio of phase transition is not as good as other state-of-the-art solvers such as probSAT and Sattime, which are the top two solvers in the random SAT category of SAT Competition 2013. Note that the major part of the random SAT benchmark in SAT Competition 2013 consists of random k-SAT instances at phase transition.

The second contribution of this paper is to improve CScoreSAT for random k-SAT instances at the threshold ratio of phase transition. The idea is to reduce the intensification of the greedy mode, because such instances have fewer models (if satisfiable). Our considerations give rise to the third scoring function dubbed $hscore_2$, which combines $score_2$ with age. This function is used to improve the greedy mode of CScoreSAT, leading to a new algorithm called HScoreSAT. In the greedy mode, HScoreSAT utilizes the $score$ property to pick the flipping variable, and breaks ties by the $hscore_2$ function. We evaluate HScoreSAT on random k-SAT ($k > 3$) instances at the threshold ratio of phase transition, including those from SAT Competition 2013, and the experimental results show that HScoreSAT significantly improves CScoreSAT on such instances.

We note that the first two functions and the CScoreSAT algorithm (Section 3), have been presented in a conference paper (Cai & Su, 2013a), while the third scoring function and the HScoreSAT algorithm (Section 4), as well as further experimental analyses (including Section 3.5 and the whole Section 5) are new contributions in this paper.

This paper proceeds as follows. Section 2 introduces some preliminary concepts. Section 3 presents the $cscore$ and $hscore$ functions and describes the CScoreSAT algorithm, along with experimental evaluations and analyses of CScoreSAT on random k-SAT ($k > 3$) instances near phase transition. Section 4 presents the $hscore_2$ function and the HScoreSAT algorithm, as well as evaluations of HScoreSAT on random k-SAT ($k > 3$) instances at phase transition and related experimental analyses. In Section 5, we study the computation of $score_2$, including its implementation, complexity and computational overhead. Finally, we give some concluding remarks and future directions in Section 6.

2. Preliminaries

In this section, we first introduce some basic definitions and notation about the problem. Then, we briefly review the notion of second level properties and related works. Finally, we introduce the configuration checking strategy, which is also an important component in our algorithms.

2.1 Basic Definitions and Notation

Given a set of n Boolean $variables$ $\{x_1, x_2, ..., x_n\}$, a $literal$ is either a variable x (which is called positive literal) or its negation $\neg x$ (which is called negative literal), and a $clause$ is a disjunction of literals. A conjunctive normal form (CNF) formula $F = c_1 \wedge c_2 \wedge ... \wedge c_m$ is a conjunction of clauses. A satisfying assignment for a formula is an assignment to its variables such that the formula evaluates to true. Given a CNF formula F, the Boolean Satisfiability problem is to find a satisfying assignment or prove that none exists.

A well-known generation model for SAT is the uniform random k-SAT model (Achlioptas, 2009). In a random k-SAT instance, each clause contains exactly k distinct non-complementary literals, and is picked up with uniform probability distribution from the set of $2^k\binom{n}{k}$ possible clauses. The clause-to-variable $ratio$ of a CNF formula F is defined as $r = m/n$, where n is the number of variables and m is the number of clauses.

For a CNF formula F, we use $V(F)$ to denote the set of all variables that appear in F. We say a variable appears in a clause, if the clause contains either x or $\neg x$. Two variables are neighbors if and only if they appear simultaneously in at least one clause. The neighbourhood of a variable x is $N(x) = \{y|y \text{ occurs in at least one clause with } x\}$, which is the set of all $neighboring\ variables$ of variable x. For a subset $X \subset V(F)$ and an assignment α, $\alpha[X]$ is the projection of α on the variables of X.

We say that a literal is true if the current value of the variable is the same as its *phase*. E.g., if x_1 = false, then the negative literal $\neg x_1$ is true, while the positive literal x_1 is not true. A clause is *satisfied* if it has at least one true literal, and *falsified* otherwise.

SLS algorithms for SAT usually select a variable to flip in each step under the guidance of $scoring\ functions$. Most SLS algorithms have more than one scoring function, and adopt one of them for the current search step according to some conditions, such as whether a local optimum is reached. A scoring function can be a simple variable property or any mathematical expression with one or more properties.

Perhaps the most popular variable property used by SLS algorithms for SAT is $score$, which measures the increase in the number of satisfied clauses by flipping a variable. The $score$ property is also defined as $score(x) = make(x) - break(x)$, where $make$ and $break$ is the number of clauses that would become satisfied and falsified, respectively, if x were to be flipped. Note that the two definitions of $score$ are equivalent. In dynamic local search algorithms which use clause weighting techniques, $score$ measures the increase in the total weight of satisfied clauses by flipping a variable, while $make$ and $break$ measures the total weight of clauses that would become satisfied and falsified, respectively, by flipping x. A variable is $decreasing$ if its $score$ is positive, and $increasing$ if its $score$ is negative. The age of a variable is defined as the number of search steps that have occurred since the variable was last flipped.

2.2 Second Level Properties

In this subsection, we introduce the second level properties, especially the second level score, which is an important concept in the proposed scoring functions in this work.

The second level properties take into account the *satisfaction degree* of clauses, which is defined as the number of true literals in the clause (Cai & Su, 2013a). A clause with a satisfaction degree of δ is said to be a δ-satisfied clause. For a variable x, $score_2(x)$ is defined as $make_2(x)$ minus $break_2(x)$, where $make_2(x)$ is the number of 1-satisfied clauses that would become 2-satisfied by flipping x, and $break_2(x)$ is the number of 2-satisfied clauses that would become 1-satisfied by flipping x. One can easily define properties of other levels and the weighted version of these properties.

The first SLS solver using second level properties is CCASat (Cai & Su, 2013b), which simply uses $score_2$ as a tie breaker and achieves surprising improvements on random k-SAT with long clauses. Then, in the conference version of this paper, we combine $score$ and $score_2$ to develop the CScoreSAT algorithm (Cai & Su, 2013a). We also propose the notion of multi-level properties and use $make_2$ to improve the famous WalkSAT/SKC algorithm (Cai, Su, & Luo, 2013a). Afterwards, multi-level $break$ is used to improve the probSAT solver (Balint, Biere, Fröhlich, & Schöning, 2014). In this work, we further exploit the $score_2$ property by using it to design scoring functions that directly guide the algorithm to pick the flipping variable.

We note that both algorithms in this work utilize the unweighted version of $score_2$ (although they use the weighted version of $score$), just as CCASat does. In our algorithms, the unweighted $score_2$ is found to be much more effective than the weighted one, yet at this time we could not figure out the reason or find an effective way using weighted $score_2$ in these algorithms.

2.3 Configuration Checking for SAT

In this subsection, we briefly introduce the configuration checking (CC) strategy for SAT, which is an important component in the proposed algorithms in this work.

Initially introduced for improving local search for the Minimum Vertex Cover (MVC) problem (Cai, Su, & Sattar, 2011), the CC strategy aims at avoiding cycling in local search, $i.e.$, revisiting the already visited candidate solutions too early. It has been successfully used in MVC (Cai et al., 2011; Cai, Su, Luo, & Sattar, 2013b), as well as SAT (Cai & Su, 2012; Luo et al., 2013; Abramé, Habet, & Toumi, 2014; Luo, Cai, Wu, & Su, 2014; Li, Huang, & Xu, 2014) and MaxSAT (Luo, Cai, Wu, Jie, & Su, 2014).

The CC strategy is based on the concept of *configuration*. One can define configuration in different ways and design different CC strategies accordingly. In the context of SAT, the configuration of a variable typically refers to truth values of all its neighboring variables (Cai & Su, 2013b). Formally, given an assignment α, the CC strategy for SAT defines the configuration $C(x_i)$ of a variable x_i as a subset of α restricted to the variables of $N(x_i)$, $i.e.$, $C(x_i) = \alpha[N(x_i)]$. If a variable in $C(x_i)$ has been flipped since the last flip of x_i then $C(x_i)$ is said changed. The CC strategy for SAT forbids the flip of a variable x_i if its configuration $C(x_i)$ has not changed since the last flip of x_i.

The CC strategy is used to decrease blind unreasonable greedy search. This strategy has been successfully applied to SAT solving, resulting in several efficient SLS algorithms for SAT, such as CCASat (Cai & Su, 2013b), Ncca+ (the bronze medal winner of the random SAT track of SAT Competition 2013) (Abramé et al., 2014), BalancedZ (Li et al., 2014) and CSCCSat (Luo et al.,

2014) (the silver and bronze medal winner of random SAT track of SAT Competition 2014), and CCAnr+glucose (Cai & Su, 2012) (the silver medal winner of hard combinatorial SAT track of SAT Competition 2014), etc.

3. Two New Scoring Functions and the CScoreSAT Algorithm

In this section, we design two new scoring functions, namely $cscore$ and $hscore$. Then we use them to develop a new SLS algorithm called CScoreSAT, which shows excellent performance on random k-SAT with $k > 3$ near the phase transition.

3.1 The $cscore$ Function

In this subsection, we introduce the $cscore$ (short for comprehensive score) function, which is a linear combination of the $score$ and $score_2$ properties.

The $score$ property characterizes the greediness of flipping a variable at the current search step, as it tends to decrease the number of falsified clauses, which is indeed the aim of the SAT problem. On the other hand, the $score_2$ property can be regarded as a measurement of look-ahead greediness, as it tends to reduce 1-satisfied clauses by transforming them into 2-satisfied clauses, noting that 1-satisfied clauses may become falsified in the next step while 2-satisfied ones do not.

It seems short sighted to simply take the $score$ property as the scoring function, especially for formulas with long clauses, in which the number of true literals varies considerably among satisfied clauses. To address this issue, we propose a scoring function that incorporates both $score$ and $score_2$. When deciding the candidate variables' priorities of being selected, although $score$ is more important than $score_2$, in some cases $score_2$ should be allowed to overwrite the priorities. For example, for two variables which have a relatively small $score$ difference and a significant $score_2$ difference, it is advisable to prefer to flip the one with greater $score_2$.

The above considerations suggest two principles in designing the desired scoring functions.

- First, the $score$ property plays a more important role;
- Second, the $score_2$ property is allowed to overwrite the variables' priorities (of being selected).

As a result, we have the notion of comprehensive score, which is formally defined as follows.

Definition 1. *For a CNF formula F, the comprehensive score function, denoted by cscore, is a function on $V(F)$ such that*

$$cscore(x) = score(x) + score_2(x)/d,$$

where d is a positive integer parameter.

Note that $cscore$ is defined to be an integer function, and thus the value of $cscore$ will be rounded down to an integer if it is not.

The $cscore$ function is a linear combination of $score$ and $score_2$ with a bias towards $score$, and thus embodies the two principles well. This function is so simple that it can be computed with little overhead and the parameter can be easily tuned. Moreover, its simplicity allows its potential usage in solving structured SAT instances and perhaps other combinatorial search problems.

Recall that a variable is decreasing if and only if it has a positive $score$. In the following, we define a new type of "deceasing" variables based on the $cscore$ function.

Definition 2. *Given a CNF formula F and its $cscore$ function, a variable x is comprehensively decreasing if and only if $cscore(x) > 0$ and $score(x) \geq 0$.*

While the condition $cscore(x) > 0$ is straightforward, the other condition $score(x) \geq 0$ requires the variable to be non-increasing. This is necessary, as flipping an increasing variable leads the local search away from the objective, which should not be accepted without any controlling mechanism such as the Metroplis probability in Simulated Annealing (Kirkpatrick, Gelatt, & Vecchi, 1983), unless the algorithm gets stuck in a local optimum.

Most SLS algorithms for SAT prefer to flip decreasing variables in the greedy search mode. In some respect, the notion of comprehensively decreasing variables is an extension of decreasing variables, and is a good alternative to be considered as flip candidates in the greedy search phases.

3.2 The $hscore$ Function

We combine $cscore$ with the diversification property age, resulting in a hybrid scoring function dubbed $hscore$, which can be used to improve the diversification mode.

One of the most commonly used variable property in the diversification mode of SLS algorithms for SAT is age. Previous SLS algorithms usually use age to pick the oldest variable from a candidate variable set (Gent & Walsh, 1993; Li & Huang, 2005; Cai & Su, 2012; Abramé et al., 2014) or only to break ties (Prestwich, 2005; Pham, Thornton, Gretton, & Sattar, 2007; Luo, Su, & Cai, 2012). In our opinion, however, these "oldest" strategies are too strict as they always prefer the oldest one, regardless of other important information such as $score$ or $cscore$. Thus, these "oldest" strategies may miss better variables quite often.

For example, suppose an SLS algorithm gets stuck in a local optimum, and it would like to pick one variable to flip from such two variables x_1 and x_2: the two variables have similar ages and x_1 is older than x_2, while $cscore(x_2)$ is significantly greater than $cscore(x_1)$. In this case, we believe x_2 is the right choice rather than the older variable x_1, as the flipping of these two variables leads to similar diversification and flipping x_2 does less harm to the object function.

Based on the above considerations, we design a hybrid scoring function taking account into both the greediness information $cscore$ and the diversification information age. The resulting scoring function is dubbed as $hscore$ and is given as follows.

Definition 3. *For a CNF formula F, the $hscore$ function is a function on $V(F)$ such that*

$$hsocre(x) = cscore(x) + age(x)/\beta = score(x) + score_2(x)/d + age(x)/\beta,$$

where d and β are positive integer parameters.

In our algorithms, when reaching a local optimum, the algorithms make use of this hybrid function. We will show that the $hscore$ function is a better choice than the "oldest" strategy for the diversification mode.

3.3 The CScoreSAT Algorithm

This section presents the CScoreSAT algorithm, which adopts the $cscore$ function to guide the search in the greedy mode, and makes use of the $hscore$ function when meets local optima.

Before getting into the details of the CScoreSAT algorithm, we first introduce two techniques employed in the algorithm.

1. **PAWS weighting scheme.** For the sake of diversification, CScoreSAT employs the PAWS clause weighting scheme (Thornton, Pham, Bain, & Ferreira Jr., 2004). Each clause is associated with a positive integer as its weight, which is initiated as 1. When a local optimum is reached, the clause weights are updated as follows. With probability sp (the so-called *smooth probability*), for each satisfied clause whose weight is larger than one, its weight is decreased by one; with probability $(1 - sp)$, the weights of all falsified clauses are increased by one.

2. **Configuration checking.** In order to reduce blind greedy search, we utilize the configuration checking strategy for SAT (Cai & Su, 2012). Recall that a variable is said to be configuration changed if and only if after its last flip at least one of its neighboring variables has been flipped. According to the configuration checking strategy, only configuration changed variables are allowed to be flipped in the greedy mode.

Algorithm 1: CScoreSAT

Input: CNF-formula F, $maxSteps$
Output: A satisfying assignment α of F, or "unknown"

begin
  $\alpha :=$ randomly generated truth assignment;
  for $step := 1$ **to** $maxSteps$ **do**
    if α *satisfies* F **then return** α;
    if $S = \{x | x$ *is comprehensively decreasing and configuration changed* $\} \neq \emptyset$ **then**
      $v :=$ a variable in S with the greatest $cscore$, breaking ties in favor of the oldest one;
    else
      update clause weights according to PAWS;
      pick a random falsified clause C;
      $v :=$ the variable in C with the greatest $hscore$;
    $\alpha := \alpha$ with v flipped;
  return "unknown";
end

The CScoreSAT algorithm is outlined in Algorithm 1, as described below. In the beginning, CScoreSAT generates a random complete assignment, initiates all clause weights to 1 and computes $score$ and $score_2$ of variables accordingly. After initialization, CScoreSAT executes a loop until it finds a satisfying assignment or reaches a limited number of steps denoted by $maxSteps$ (or a given cutoff time).

Like most SLS algorithms for SAT, CScoreSAT switches between two modes. In each search step, it works in either the greedy mode or the diversification mode, depending on whether there exist comprehensively decreasing variables that are configuration changed. If there exist such variables, CScoreSAT works in the greedy mode. It picks such a variable with the greatest $cscore$ value to flip, breaking ties by preferring the oldest one.

If no variables are both comprehensively decreasing and configuration changed, then CScoreSAT switches to the diversification mode. It first updates clause weights according to the

PAWS scheme. Then it randomly selects a falsified clause C, and picks the variable from C with the greatest $hscore$ value to flip.

3.4 Evaluations of CScoreSAT

In this subsection, we carry out extensive experiments to evaluate CScoreSAT on random k-SAT instances with $k \in \{4, 5, 6, 7\}$ near phase transition. First, we compare CScoreSAT with state-of-the-art SLS solvers on random 5-SAT and 7-SAT instances. Then, we compare CScoreSAT with state-of-the-art SLS solvers on random k-SAT with instances $k \in \{4, 5, 6, 7\}$ from SAT Challenge 2012. Finally, we study the effectiveness of the $cscore$ and $hscore$ functions through empirical analysis on random 5-SAT and 7-SAT instances.

3.4.1 BENCHMARKS AND EXPERIMENT PRELIMINARIES

All the instances used in these experiments are generated according to the random k-SAT model near the solubility phase transition. Specifically, we adopt the following five benchmarks. The first two benchmarks are for random 5-SAT, and the third and fourth benchmarks are for random 7-SAT, while the last one consists of random k-SAT instances with $k = 4, 5, 6, 7$ at various ratios.

1. **5-SAT Comp11:** all large random 5-SAT instances from SAT Competition 2011 ($r = 20$, $750 \leq n \leq 2000$, 50 instances, 10 for each size).

2. **5-SAT Huge:** 5-SAT instances generated randomly according to the random k-SAT model ($r = 20$, $3000 \leq n \leq 5000$, 500 instances, 100 for each size).

3. **7-SAT Comp11:** all large random 7-SAT instances from SAT Competition 2011 ($r = 85$, $150 \leq n \leq 400$, 50 instances, 10 for each size).

4. **7-SAT Random:** 7-SAT instances generated randomly according to the random k-SAT model ($r = 85$, $220 \leq n \leq 300$, 500 instances, 100 for each size).

5. **SAT Challenge 2012:** all random k-SAT instances with $k > 3$ from SAT Challenge 2012 (480 instances, 120 for each k-SAT, $k = 4, 5, 6, 7$), which vary in both size and ratio. These random instances occupy 80% of the random benchmark in SAT Challenge 2012, indicating that the importance of random k-SAT instances with $k > 3$ has been highly recognized by the SAT community. The instances vary from 800 variables at $r = 9.931$ to 10000 variables at $r = 9.0$ for 4-SAT, from 300 variables at $r = 21.117$ to 1600 variables at $r = 20$ for 5-SAT, from 200 variables at $r = 43.37$ to 400 variables at $\alpha = 40$ for 6-SAT, and from 100 variables at $r = 87.79$ to 200 variables at $r = 85$ for 7-SAT.

parameter	4-SAT	5-SAT	6-SAT	7-SAT
sp (for PAWS)	0.62	0.62	0.9	0.9
d	9	8	7	6
β	2000	2000	2000	2000

Table 1: Parameter setting of CScoreSAT

CScoreSAT is implemented in C++ and compiled by g++ with the '-O2' option. The parameter setting of CScoreSAT is reported in Table 1. We compare CScoreSAT with four state-of-the-art SLS solvers, including `Sparrow2011` (Balint & Fröhlich, 2010), `CCASat` (Cai & Su, 2013b), `probSAT` (Balint & Schöning, 2012), and `Sattime2012` (Li & Li, 2012). Sparrow2011 and probSAT won the gold medal of the random SAT track of the SAT competitions 2011 and 2013 respectively. CCASat is the winner of this same category in SAT Challenge 2012. Sattime regularly won medals during SAT competitions of the same track.

All experiments are carried out parallel on a workstation under a 32-bit Ubuntu Linux Operation System, using 2 cores of Intel(R) Core(TM) 2.6 GHz CPU and 8 GB RAM. The experiments are conducted with EDACC, an experimental platform for testing SAT solvers, which has been used for SAT Challenge 2012 and SAT Competition 2013. Each run terminates upon either finding a satisfying assignment or reaching a given cutoff time which is set to 5000 seconds (as in SAT Competition 2011) for the 5-SAT and 7-SAT benchmarks, and 1000 seconds for the SAT Challenge 2012 benchmark (close to the cutoff in SAT Challenge 2012, $i.e.$, 900 seconds).

For the 5-SAT Comp11 and 7-SAT Comp11 benchmarks (where each instance class has 10 instances), we run each solver 10 times for each instance and thus 100 runs for each instance class. For the 5-SAT Huge and 7-SAT Random benchmarks (where each instance class contains 100 instances) and the SAT Challenge 2012 benchmark (120 k-SAT instances for each k), we run each solver one time for each instance, as the instances in each class are enough to test the performance of the solvers.

For each solver on each instance class, we report the number of successful runs in which a satisfying assignment is found ("suc runs") or the solved instances ("#solved"), as well as the PAR10 ("par10"), which is a penalized average run time where a timeout of a solver is penalized as 10$*$(cutoff time). Note that PAR10 is adopted in SAT competitions and has been widely used in the literature as a prominent performance measure for SLS-based SAT solvers (KhudaBukhsh, Xu, Hoos, & Leyton-Brown, 2009; Tompkins & Hoos, 2010; Tompkins, Balint, & Hoos, 2011; Balint & Schöning, 2012). The results in **bold** indicate the best performance for an instance class. If a solver has no successful run on an instance class, the corresponding "par10" is marked with "n/a".

3.4.2 EXPERIMENTAL RESULTS OF CSCORESAT

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

Results on 5-SAT Comp11 Benchmark:

Table 2 shows the comparative results on the 5-SAT Comp11 benchmark. As is clear from Table 2, CScoreSAT shows significantly better performance than other solvers on the whole benchmark. CScoreSAT is the only solver that solves all these 5-SAT instances in all runs. Also, CScoreSAT significantly outperforms its competitors in terms of run time, which is more obvious as the instance size increases. In particular, on the 5-SAT-v2000 instances, which are of the largest size in SAT competitions, the runtime of CScoreSAT is 15 times less than that of CCASat, and 2 orders of magnitudes less than that of other state-of-the-art SLS solvers.

Results on 5-SAT Huge Benchmark:

The experimental results on the 5-SAT Huge benchmark are presented in Table 3. It is encouraging to see the performance of CScoreSAT remains surprisingly good on these very large 5-SAT instances, where state-of-the-art solvers show very poor performance. CScoreSAT solves these

Instance Class	Sattime2012 suc runs par10	Sparrow2011 suc runs par10	probSAT suc runs par10	CCASat suc runs par10	CScoreSAT suc runs par10
5-SAT-v750	100 754	100 51	100 88	100 47	**100** **35**
5-SAT-v1000	100 1254	100 159	100 185	100 81	**100** **38**
5-SAT-v1250	95 5288	100 174	100 237	100 128	**100** **47**
5-SAT-v1500	56 24101	99 1231	98 1753	100 443	**100** **145**
5-SAT-v2000	14 43249	72 15288	71 15635	93 4386	**100** **289**

Table 2: **Experimental results on the 5-SAT Comp11 benchmark.** There are 10 instances in each class and each solver is executed 10 times on each instance with a cutoff time of 5000 seconds.

Instance Class	Sattime2012 suc runs par10	Sparrow2011 suc runs par10	probSAT suc runs par10	CCASat suc runs par10	CScoreSAT suc runs par10
5-SAT-v3000	0 n/a	31 35360	40 30867	64 19403	**100** **694**
5-SAT-v3500	0 n/a	8 46147	6 47188	35 33540	**100** **1431**
5-SAT-v4000	0 n/a	4 48080	3 48591	10 45287	**87** **8167**
5-SAT-v4500	0 n/a	0 n/a	0 n/a	0 n/a	**62** **21513**
5-SAT-v5000	0 n/a	0 n/a	0 n/a	0 n/a	**38** **32005**

Table 3: **Experimental results on the 5-SAT Huge benchmark.** There are 100 instances in each class and each solver is executed one time on each instance with a cutoff time of 5000 seconds.

5-SAT instances with up to (at least) 3500 variables consistently ($i.e.$, with 100% success rate), and is about 30 times faster than other solvers on the 5-SAT-v3500 instances. Furthermore, CScoreSAT succeeds in 62 and 38 runs for the 5-SAT-v4500 and 5-SAT-v5000 instances respectively, whereas all its competitors fail to find a solution for any of these instances. Indeed, to the best of our knowledge, such large random 5-SAT instances (at $r = 20$) are solved for the first time. Given the good performance of CScoreSAT on the 5-SAT instances with 5000 variables, we are confident it could be able to solve larger 5-SAT instances.

Instance Class	Sattime2012 suc runs par10	Sparrow2011 suc runs par10	probSAT suc runs par10	CCASat suc runs par10	CScoreSAT suc runs par10
7-SAT-v150	100 498	100 642	88 6980	100 232	**100** **131**
7-SAT-v200	49 26998	17 41912	11 44806	72 14912	**90** **5853**
7-SAT-v250	2 49095	0 n/a	0 n/a	7 46731	**35** **34070**
7-SAT-v300	0 n/a	0 n/a	0 n/a	0 n/a	**11** **44776**
7-SAT-v400	0 n/a	0 n/a	0 n/a	0 n/a	0 n/a

Table 4: **Experimental results on the 7-SAT Comp11 benchmark.** There are 10 instances in each class and each solver is executed 10 times on each instance with a cutoff time of 5000 seconds.

Instance Class	Sattime2012 suc runs par10	Sparrow2011 suc runs par10	probSAT suc runs par10	CCASat suc runs par10	CScoreSAT suc runs par10
7-SAT-v220	39 31868	13 43407	10 45253	68 17189	**83** **10639**
7-SAT-v240	13 43935	2 49051	2 49052	33 34158	**66** **17901**
7-SAT-v260	4 48113	0 n/a	0 n/a	9 45736	**53** **24825**
7-SAT-v280	0 n/a	0 n/a	0 n/a	5 47605	**24** **39283**
7-SAT-v300	0 n/a	0 n/a	0 n/a	0 n/a	**11** **44889**

Table 5: **Experimental results on the 7-SAT Random benchmark.** There are 100 instances in each class and each solver is executed one time on each instance with a cutoff time of 5000 seconds.

Results on 7-SAT Comp11 Benchmark:

Table 4 summarizes the experimental results on the 7-SAT Comp11 benchmark. None of the solvers can solve any 7-SAT instance with 400 variables, indicating that random 7-SAT instances near the phase transition are so difficult even with a relatively small size. Nevertheless, CScoreSAT significantly outperforms its competitors on this 7-SAT benchmark, and is the only solver that can solve such 7-SAT instances with 300 variables. Actually, all the competitors become ineffective

Instance Class	Sattime2012 #solved par10	Sparrow2011 #solved par10	probSAT #solved par10	CCASat #solved par10	CScoreSAT #solved par10
4-SAT	49 6031	79 3514	111 778	112 751	**119** **174**
5-SAT	32 7407	52 5812	54 5657	71 4264	**84** **3146**
6-SAT	84 3187	72 4193	76 3877	99 1887	**110** **935**
7-SAT	81 3422	65 4714	57 5380	77 3734	**91** **2559**
Over All	246 5011	268 4558	298 3923	359 2659	**404** **1703**

Table 6: **Experimental results on SAT Challenge 2012 benchmark.** Each instance class contains 120 instances, and each solver is executed once on each instance with a cutoff time of 1000 seconds.

(among which CCASat has the highest success rate of 7%) on the 7-SAT-v250 instances, while CScoreSAT still achieves a success rate of 35% for this instance class.

Results on 7-SAT Random Benchmark:

The sizes of random 7-SAT instances from SAT Competition 2011 are not continuous enough to provide a good spectrum of instances for SLS solvers. In order to investigate the detailed performance of CScoreSAT and state-of-the-art SLS solvers on random 7-SAT instances, we evaluate them on the 7-SAT Random benchmark, where the instance size increases more slowly. Once again, Table 5 suggests that the difficulty of such 7-SAT instances increases significantly with a relatively small increment of the size. As reported in Table 5, the results show CScoreSAT dramatically outperforms its competitors. Compared to the competitors whose performance descends steeply as the instance size increases, CScoreSAT shows good scalability. For example, from 7-SAT-v220 to 7-SAT-v260, the success rates of all the competitors decline eight times or more, whereas that of CScoreSAT drops only thirty percents. When coming to the 7-SAT-v260 instances, probSAT and Sparrow2011 fail in all runs, and the other competitors succeed in less than 10 runs, while CScoreSAT succeeds in 53 runs. Finally, CScoreSAT is the only solver that survives throughout the whole benchmark.

Results on SAT Challenge 2012 Benchmark:

To investigate the performance of CScoreSAT on random k-SAT instances with various k ($k > 3$), we compare it with state-of-the-art solvers on all random k-SAT instances with $k > 3$ from SAT Challenge 2012. Table 6 reports the number of solved instances and PAR10 for each solver on each k-SAT instance class. The results show that CScoreSAT significantly outperforms its competitors in terms of both metrics. Overall, CScoreSAT solves 404 instances. Further observations show that CScoreSAT solves 365 instances within half cutoff time, whereas none of its competitors solves more than 360 instances within the cutoff time. More encouragingly, Table 6 shows that CScoreSAT solves the most k-SAT instances for each k, which illustrates its robustness. The good performance

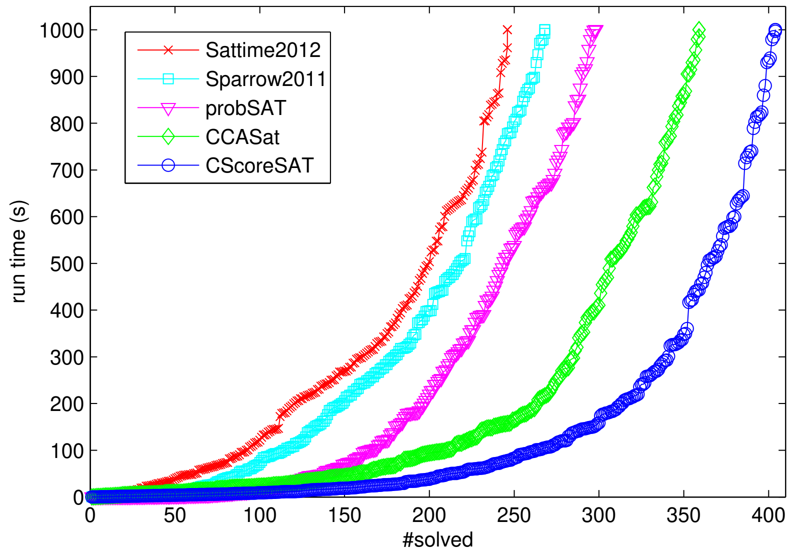

Figure 1: **Comparison of run time distributions on the SAT Challenge 2012 benchmark, with a cutoff time of 1000 seconds.**

of CScoreSAT on the SAT Challenge 2012 benchmark is also clearly illustrated by Figure 1, which summarizes the run time distributions of the solvers on this benchmark.

3.5 Experimental Analyses of $cscore$ and $hscore$ Functions

In order to demonstrate the effectiveness of the $cscore$ and $hscore$ functions, we also test two alternative versions of CScoreSAT, namely CScoreSAT$_1$ and CScoreSAT$_2$. These two algorithms are modified from CScoreSAT as follows.

- CScoreSAT$_1$: in the greedy mode, CScoreSAT$_1$ uses $score$ as the scoring function instead of $cscore$; also, CScoreSAT$_1$ does not utilize the concept of comprehensively decreasing, and a variable is allowed to flip if it is decreasing and configuration changed.

- CScoreSAT$_2$: in the diversification mode, CScoreSAT$_2$ uses the age property instead of $hscore$ as the scoring fucntion, $i.e.$, it picks the oldest variable from the selected falsified clause.

We carry out experiments to compare CScoreSAT with its two degraded versions on random 5-SAT and 7-SAT instances. The experimental results are reported in Table 7. An obvious observation is that the performance of CScoreSAT$_1$ is essentially worse than that of CScoreSAT. For example, it cannot solve any 5-SAT instance with 2000 variables or any 7-SAT instance with 250 instance. This indicates the $cscore$ function is critical to the good performance of CScoreSAT. Compared to $cscore$, the $hscore$ function used in the random mode does not show that much contribution. Nevertheless, the usage of $hscore$ does improve CScoreSAT's performance on 5-SAT and 7-SAT instances. A more careful comparison of CScoreSAT and CScoreSAT$_2$ shows that $hscore$ is more important in solving 7-SAT instances than 5-SAT ones.

Instance Class	CScoreSAT_1 suc runs par10	CScoreSAT_2 suc runs par10	CScoreSAT suc runs par10
5-SAT-v1500	41 31452	100 152	**100** **145**
5-SAT-v2000	0 n/a	100 330	**100** **289**
5-SAT-v4000	0 n/a	78 12118	**87** **8167**
7-SAT-v150	89 5359	100 569	**100** **131**
7-SAT-v200	22 40406	75 13669	**90** **5853**
7-SAT-v250	0 n/a	10 45329	**35** **34070**

Table 7: **Comparison of CScoreSAT and its two alternative algorithms on random 5-SAT and 7-SAT instances.**

4. Improving CScoreSAT on Random k-SAT at Phase Transition

The above section shows the excellent performance of CScoreSAT on random k-SAT ($k > 3$) near phase transition. However, the performance of CScoreSAT degrades on those instances at phase transition. CScoreSAT participated in the satisfiable random category of SAT Competition 2013, where the major part of the benchmark consists of instances generated at the threshold ratio of phase transition. Although it is ranked 4th in the category, its performance is not good enough on this kind of instances, and is worse than other state-of-the-art SLS solvers such as probSAT and Sattime2013, which are the top two solvers in the satisfiable random category of SAT Competition 2013.

This section improves CScoreSAT for random k-SAT ($k > 3$) at phase transition. To this end, we propose another scoring function combining $score_2$ and age, and utilize it to improve the greedy mode of CScoreSAT, resulting in a new algorithm called HScoreSAT. Our experiments show that HScoreSAT significantly improves CScoreSAT and gives state-of-the-art performance on random k-SAT ($k > 3$) at the threshold ratio of phase transition. We also compare CScoreSAT and HScoreSAT on instances with various ratios and find the boundary ratios beyond which HScoreSAT outperforms CScoreSAT.

4.1 The $hscore_2$ Function and the HScoreSAT Algorithm

An important issue in SLS algorithms for SAT is the balance between intensification and diversification. Indeed, most improvements on SLS algorithms for SAT are due to proper regulation of intensification and diversification in local search. For random k-SAT instances at the solubility phase transition, most of the search regions do not contain a model (if the instance is satisfiable). Therefore, it is inadvisable to have strong intensification for such instances, which might waste the search much time on unpromising regions so that the search does not explore enough regions for discovering a model.

In order to improve CScoreSAT for random k-SAT instances at phase transition, we propose to reduce intensification in the greedy mode. In CScoreSAT, we use $cscore$ as the scoring function, and break ties by age. As mentioned before, the $cscore$ function is quite a greedy scoring function as it combines $score$ and $score_2$, which represent the greediness and look-ahead greediness respectively. Therefore, in our opinion, $cscore$ is not suitable for random k-SAT instances at phase transition.

Recalling that the object of SLS algorithms for SAT is to minimize the number or total weight of falsified clauses, the $score$ property should be the primary criterion in the greedy mode. Also, we believe the $score_2$ property is important information for solving long-clause instances, as it considers the satisfaction degree of clauses. However, when $score$ and $score_2$ are combined together as the primary scoring function, it is too intensifying for solving (satisfiable) random k-SAT instances at phase transition.

Based on the above considerations, we move $score_2$ from the primary scoring function to the tie-breaking function, where it is combined with the diversification property age. This leads to a new scoring function which we refer to as $hscore_2$ as it is a hybrid function of $score_2$ and age.

Definition 4. *For a CNF formula F, the $hscore_2$ function is a function on $V(F)$ such that*

$$hscore_2(x) = score_2(x) + age(x)/\theta,$$

where θ is a positive integer parameter.

Accordingly, we modify the greedy mode of CScoreSAT, and obtain a new algorithm which we refer to as HScoreSAT. The pseudo-codes of HScoreSAT is given in Algorithm 2.

HScoreSAT differs from CScoreSAT in the following two aspects. First, although both algorithms utilize the CC strategy, HScoreSAT only allows decreasing variables to be flipped in the greedy mode, while CScoreSAT allows comprehensively decreasing variables (which is a super-set of decreasing variables) to be flipped. More importantly, HScoreSAT uses $hscore_2$ to break ties in the greedy mode, while CScoreSAT breaks ties by age.

Since the $hscore_2$-based tie-breaking is an important component of HScoreSAT, we are interested in this question: *when there exist configuration changed decreasing variables, how often the tie-breaking is executed to pick one from them?* We have conducted an experiment on the threshold benchmark from the random satisfiable category of SAT Competition 2013 to calculate this frequency, which is the ratio of the following two statistics.

- #steps$_{ccd}$: the number of steps in which configuration changed decreasing (CCD) variables exist.

- #steps$_{bt}$: the number of steps in which configuration changed decreasing (CCD) variables exist, and the best CCD variable is picked via $hscore_2$-based tie-breaking.

The experimental results are summarized in Table 8, which are averaged over all instances with each run per instance. As is demonstrated in Table 8, the frequency of the $hscore_2$-based tie-breaking in CCD steps is significant, and is very high for 4-SAT and 5-SAT (68% and 60% respectively). This indicates that the $hscore_2$-based tie-breaking mechanism plays a critical role in HScoreSAT. Another interesting observation is that this frequency decreases with the length of clauses in the instance.

Algorithm 2: HScoreSAT

```
Input: CNF-formula F, maxSteps
Output: A satisfying assignment α of F, or "unknown"
begin
    α := randomly generated truth assignment;
    for step := 1 to maxSteps do
        if α satisfies F then return α;
        if S = {x|x is decreasing and configuration changed } ≠ ∅ then
            v := a variable in S with the greatest score, breaking ties by preferring the one
            with the greatest hscore_2;
        else
            update clause weights according to PAWS;
            pick a random falsified clause C;
            v := a variable in C with the greatest hscore;
        α := α with v flipped;
    return "unknown";
end
```

	4-SAT	5-SAT	6-SAT	7-SAT
#steps$_{bt}$	218465135	514417498	198497368	17260095
#steps$_{ccd}$	317682739	849513823	404629074	43678889
$\frac{\#steps_{bt}}{\#steps_{ccd}}$	68%	60%	49%	38%

Table 8: Averaged number of CCD steps and $hscore_2$-based tie-breaking steps, as well as their averaged ratio for each k-SAT with $k \in \{4, 5, 6, 7\}$ in the threshold benchmark from SAT Competition 2013.

4.2 Evaluations of HScoreSAT on Threshold Instances

In this subsection, we carry out extensive experiments to evaluate HScoreSAT on random k-SAT instances with $k \in \{4, 5, 6, 7\}$ at phase transition. First, we compare HScoreSAT with CScoreSAT as well as state-of-the-art SLS solvers on the random benchmark at the threshold of phase transition from SAT Competition 2013. Then, we compare HScoreSAT with state-of-the-art SLS solvers on large-sized random k-SAT ($k \in \{4, 5, 6, 7\}$) instances generated randomly at the threshold of phase transition.

4.2.1 BENCHMARK AND EXPERIMENT PRELIMINARIES

In the experiments in this section, all benchmark instances are generated according to the random k-SAT model at the threshold ratio of the solubility phase transition. These instances have a clause-to-variable ratio equal to the conjectured threshold ratio of the solubility phase transition[2] (Mertens, Mézard, & Zecchina, 2006). Specifically, we adopt the following two benchmarks.

2. The clause-to-variable ratio for which 50% of the uniform random formulas are satisfiable. For most algorithms, the closer a formula is generated near the threshold ratio, the harder it is to solve it.

1. **Threshold Comp13:** the threshold benchmark from the random satisfiable category of SAT Competition 2013. For each k-SAT, the instances have various sizes. We also note that no filtering was applied to construct the competition suite. As a consequence, a significant fraction (approximately 50%) of the generated threshold instances is unsatisfiable. The details of the benchmark are given in Table 9.

2. **Large-sized Threshold:** random k-SAT instances at the threshold ratio of phase transition, generated randomly by the random k-SAT generator[3] used in SAT Competition 2013. This benchmark contains 400 instances, 100 for each k-SAT class with $k \in \{4, 5, 6, 7\}$. The sizes of instances in this benchmark (n = 2000, 550, 250, 150 for k = 4, 5, 6, 7, respectively) are relatively large compared to those in the Threshold Comp13 benchmark. These instances are evenly divided into two categories: the `training set` and `test set`, both of which have 50 instances for each k-SAT class.

Note that the `training set` is only used to tune the parameters in HScoreSAT, and then HScoreSAT with the tuned parameter setting is evaluated on Threshold Comp13 benchmark and the `test set` in Large-sized Threshold benchmark.

	4-SAT	5-SAT	6-SAT	7-SAT
#inst.	50	50	50	50
ratio	9.931	21.117	43.37	87.79
size	$n \in \{830, 860, ..., 2300\}$	$n \in \{305, 310, ..., 550\}$	$n \in \{191, 192, ..., 240\}$	$n \in \{91, 92, ..., 140\}$

Table 9: The instance numbers, ratios and sizes for each k-SAT with $k \in \{4, 5, 6, 7\}$ in the Threshold Comp13 benchmark.

HScoreSAT is implemented on the basis of CScoreSAT source code and complied by g++ with the '-O2' option. The parameter setting of HScoreSAT is presented in Table 10, which are tuned based on the `training set` of the Large-sized Threshold benchmark. We compare HScoreSAT with CScoreSAT, as well as three other state-of-the-art SLS solvers, including `CCASat`, `probSAT` (version 2013) and `Sattime2013`. Especially, we note that probSAT and Sattime2013 are the top two solvers in the random SAT track in SAT Competition 2013.

parameter	4-SAT	5-SAT	6-SAT	7-SAT
sp	0.75	0.75	0.92	0.9
d	9	8	7	6
θ	50	100	500	500
β	500	500	500	500

Table 10: Parameter setting of HScoreSAT

The computing environments for these experiments are the same as those used for experiments in Section 3. Following the experiment setup in SAT Competition 2013, we perform each solver one run on each instance, where each run terminates upon either finding a satisfying assignment or reaching a given cutoff time which is set to 5000 seconds. We report the number of solved instances

3. http://sourceforge.net/projects/ksatgenerator/

Instance Class	CCASat #solved par10	Sattime2013 #solved par10	probSAT #solved par10	CScoreSAT #solved par10	HScoreSAT #solved par10
4-SAT	10	8	13	9	**14**
($r = 9.931$)	40168	42174	37160	41193	**36176**
5-SAT	9	9	10	8	**11**
($r = 21.117$)	41117	41153	40185	42176	**39296**
6-SAT	15	18	15	13	**19**
($r = 43.37$)	35435	31445	35440	37211	**31538**
7-SAT	23	**26**	23	21	25
($r = 87.79$)	27308	**24351**	27363	29189	25275
Over All	57	61	61	51	**69**
	36007	35006	35037	37442	**33071**

Table 11: **Experimental results on Threshold Comp13 benchmark.** Each instance class contains 50 instances and each solver is executed once on each instance with a cutoff time of 5000 seconds.

("#solved") and PAR10 for each k-SAT class and the whole benchmark (as in the competition). The rules at SAT competitions establish that the winner is the solver which solves the most instances, and ties are broken by selecting the solver with the minimum PAR10.

4.2.2 EXPERIMENTAL RESULTS ON THRESHOLD BENCHMARK

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

Results on Threshold Comp13 Benchmark:

Table 11 presents the experimental results of HScoreSAT and its competitors on random k-SAT instances at phase transition from SAT Competition 2013[4]. Since HScoreSAT is based on CScoreSAT, we first compare these two solvers. As shown in Table 11, HSocreSAT solves more instances than CScoreSAT on all instance classes. Overall, CScoreSAT solves 51 instances, while HScoreSAT solves 69 instances, which is 1.35 times as many as CScoreSAT does.

HScoreSAT solves a few more instances than probSAT and Sattime2013. Overall, HScoreSAT solves 69 instances, compared to 61 for both probSAT and Sattime2013 and 57 for CCASat. Further observation shows that, HScoreSAT has similar performance with probSAT on random 4-SAT and 5-SAT instances, and has similar performance with Sattime2013 on 6-SAT and 7-SAT instances.

Results on Large-sized Threshold Benchmark:

To mesure the performance of HScoreSAT on random phase-transition k-SAT instances more accurately, we additionally test HScoreSAT on the `test set` of the Large-sized Threshold benchmark, compared with Sattime2013 and probSAT, which are the top two solvers in the random SAT track in SAT Competition 2013.

4. It seems that our machine is slightly slower than the ones used in SAT Competition 2013, as Sattime2013, probSAT and CScoreSAT all solved slightly fewer instances in our experiment than they did in the competition. CCASat did not participate in SAT Competition 2013.

The results are presented in Table 12. For 4-SAT class, HScoreSAT and probSAT solve the same number of instances, but HScoreSAT has less accumulative run time. For 5-SAT and 6-SAT classes, HScoreSAT solves the most instances. Particularly, HScoreSAT shows significantly superior performance than other solvers on 6-SAT class, where it solves 9 instances, while Sattime2013 and probSAT both solve 4 instances. The only instance class for which HScoreSAT does not give the best performance is 7-SAT. Nevertheless, on this instance class, HScreSAT has similar performance as the best solver Sattime2013, solving only one less instance. For the whole benchmark, HScoreSAT solves 40 instances, compared to 26 and 28 instances for Sattime2013 and probSAT.

Instance Class	Sattime2013 #solved par10	probSAT #solved par10	HScoreSAT #solved par10
4-SAT-v2000 ($r = 9.931$)	0 n/a	8 42197	8 **42181**
5-SAT-v550 ($r = 21.117$)	8 42147	9 41262	**10** **40130**
6-SAT-v300 ($r = 43.37$)	4 46120	4 46132	**9** **41523**
7-SAT-v150 ($r = 87.79$)	**14** **36433**	7 43248	13 37091
Over All	26 43675	28 43209	**40** **40231**

Table 12: **Experimental results on the Large-size Threshold benchmark.** Each instance class contains 50 instances and each solver is executed once on each instance with a cutoff time of 5000 seconds.

4.3 Experimental Analyses of the $hscore_2$ Function

To demonstrate the effectiveness of the $hscore_2$ function, we test two alternative versions of HScoreSAT. These two algorithms are different from HScoreSAT only in the tie-breaking mechanism in the greedy mode.

- HScoreSAT$_1$ breaks ties in the greedy mode by preferring the variable with the greatest $score_2$;
- HScoreSAT$_2$ breaks ties in the greedy mode by preferring the variable with the greatest age.

The comparative results of HScoreSAT and its two alternative versions are displayed in Table 13. It is clear from the table that HScoreSAT has superior performance than both its alternatives on all instance classes. More careful observations show that for 4-SAT and 5-SAT instances, the performance of HScoreSAT$_2$ and HScoreSAT are similar, which are significantly better than that of HScoreSAT$_1$. This indicates that the age property is more suitable than $score_2$ as a tie-breaker for 4-SAT and 5-SAT, and is almost as good as the $hscore_2$ for tie-breaking. In contrast, the performance

Instance Class	HScoreSAT_1 #solved par10	HScoreSAT_2 #solved par10	HScoreSAT #solved par10
4-SAT ($r = 9.931$)	5 45062	13 37242	**14** **36176**
5-SAT ($r = 21.117$)	7 43136	10 40293	**11** **39296**
6-SAT ($r = 43.37$)	16 34255	16 34371	**19** **31538**
7-SAT ($r = 87.79$)	24 26328	22 28128	**25** **25275**
Over All	52 37195	61 35008	**69** **33071**

Table 13: **Comparative results of HScoreSAT and its two alternative solvers on the Threshold Comp13 benchmark.** Each solver is executed one time on each instance, with a cutoff time of 5000 seconds.

of HScoreSAT_1 and HScoreSAT on 7-SAT are similar, which are better than that of HScoreSAT_2. This indicates that $score_2$ is more suitable than age as a tie-breaker for 7-SAT. Indeed, it is from this experimental analysis that we gain the intuition to set θ in the $hscore_2$ function to be a relatively small value for 4-SAT and 5-SAT instances, and a relatively large value for 7-SAT. However, for 6-SAT, both alternatives cannot achieve performance close to that of HScoreSAT.

4.4 Evaluation on Huge Random k-SAT in SAT Competition 2013

In the random SAT category of SAT Competition 2013, there are two kinds of instances. Besides the instances at phase-transition threshold, there are also instances whose ratios are not that close to phase transition while at the same time they have huge sizes. In this subsection, we conduct experiments to evaluate the performance of our solvers on these huge instances, compared with state-of-the-art solvers.

The experimental results are presented in Table 14, which show that CScoreSAT is clearly the best solver on this benchmark of huge instances. CScoreSAT gives the best performance for all k-SAT instance classes except for 4-SAT, and especially it solves more 6-SAT and 7-SAT instances than all other solvers. For 4-SAT, CScoreSAT solves as many instances as probSAT but the PAR10 is a little more than probSAT's. These experimental results confirm the good performance of CScoreSAT on the huge benchmark in SAT Competition 2013, where it also solved more huge instances than probSAT and Sattime2013.

4.5 Boundary Ratios for Performance of CScoreSAT and HScoreSAT

As we mentioned before, CScoreSAT has good performance on solving random k-SAT ($k > 3$) instances at some ratios near the phase-transition threshold, such as the ratios of large random instances in SAT Competition 2011. On the other hand, HScoreSAT is improved from CScoreSAT for solving random k-SAT ($k > 3$) instances at the phase-transition threshold. Thus, we conjecture

Instance Class	CCASat #solved par10	Sattime2013 #solved par10	probSAT #solved par10	CScoreSAT #solved par10	HScoreSAT #solved par10
4-SAT-v500000	4	3	5	5	5
($r \in [7.0, 9.5]$)	17017	25205	**8385**	9096	9418
5-SAT-v250000	3	2	**4**	**4**	3
($r \in [15.0, 20.0]$)	25106	33502	16710	**16535**	25116
6-SAT-v100000	2	2	2	**3**	2
($r \in [30.0, 40.0]$)	33386	33710	33338	**25200**	33376
7-SAT-v50000	1	1	1	**2**	1
($r \in [60.0, 85.0]$)	41693	41963	41669	**34167**	41683
Over All	10	8	12	**14**	11
	29300	33595	25026	**21249**	27398

Table 14: **Experimental results on huge random k-SAT ($k > 3$) instances from SAT Competition 2013.** Each instance class contains 6 instances and each solver is executed once on each instance with a cutoff time of 5000 seconds.

there exists a boundary ratio for each k-SAT, beyond which HScoreSAT outperforms CScoreSAT. This subsection is dedicated to finding these boundary ratios through experiments.

Our experiment is carried out on SAT Challenge 2012 benchmark, where each k-SAT has 10 different ratios and there are 12 instances for each ratio. Details of the bechmark can be found in its benchmark description. We run CScoreSAT and HScoreSAT one time on each instance with a cutoff time of 1000 seconds, and compare the performance of the two solvers at each ratio.

The comparison results are presented in Table 15. The results suggest that there exists a boundary ratio r^*, beyond which HScoreSAT gives better performance than CScoreSAT. This is especially clear for 4-, 5- and 7-SAT, while not so clear for 6-SAT as HScoreSAT solves more instances than CScoreSAT at all rations which are not smaller than 42.359 except for one ratio $r = 42.696$, where CScoreSAT solves one more instance. To check whether this result is just an outlier due to a single instance, we conduct an additional experiment to execute both solvers 10 times on each instance at $r = 42.696$. The experimental results show that the two solvers have very close performance — HScoreSAT succeeds in 84 runs while CScoreSAT succeeds in 83 runs. We give the conjectured interval (r^*_{min}, r^*_{max}) of the boundary ratio r^* for each k-SAT in Table 16.

These results suggest that, a hybrid solver combining CScoreSAT and HScoreSAT would have good performance on k-SAT instances with long clauses at different ratios. However, both CScoreSAT and HScoreSAT have poor performance on random 3-SAT instances. Hence, these two solvers or their hybrid solver did not participate in SAT Competition 2014, as the competition requires a participating solver to be a core solver which can have at most two different solvers. Instead, we develop a solver called CSCCSat, which combines two solvers namely FrwCB (for large sized instances) (Luo et al., 2014) and DCCASat (for threshold instances) (Luo et al., 2014). Note that DCCASat is improved from HScoreSAT by using the double configuration checking heuristic, and it also uses the $hscore$ and $hscore_2$ functions for random k-SAT with $k > 3$. In SAT Competition 2014, CSCCSat won the bronze medal of random SAT track, and especially it gives the best performance for threshold instances, indicating the effectiveness of our scoring functions.

	CScoreSAT HScoreSAT		CScoreSAT HScoreSAT		CScoreSAT HScoreSAT		CScoreSAT HScoreSAT
4-SAT r=9	**4.8**(12) 6.5(12)	5-SAT r=20	**222(12)** 1060(11)	6-SAT r=40	**4.1**(12) 11.5(12)	7-SAT r=85	**4345(7)** 8346(2)
4-SAT r=9.121	**9.7**(12) 10.8(12)	5-SAT r=20.155	**271(12)** 2022(10)	6-SAT r=40.674	**8.6**(12) 36.3(12)	7-SAT r=85.558	**2767(9)** 6773(4)
4-SAT r=9.223	**11.6**(12) 18.6(12)	5-SAT r=20.275	**1918(10)** 3471(8)	6-SAT r=41.011	**29.1**(12) 52.3(12)	7-SAT r=85.837	**2666(9)** 5065(6)
4-SAT r=9.324	**19.1**(12) 29.1(12)	5-SAT r=20.395	**2935(9)** 6025(5)	6-SAT r=41.348	**76.6**(12) 98.4(12)	7-SAT r=86.116	**3435(8)** 4296(7)
4-SAT r=9.425	**42.7**(12) 63.8(12)	5-SAT r=20.516	**6108(5)** 7647(3)	6-SAT r=41.685	**83(12)** 1827(10)	7-SAT r=86.395	**1902(10)** 2797(9)
4-SAT r=9.526	**66.5**(12) 72.2(12)	5-SAT r=20.636	**4363(7)** 6003(5)	6-SAT r=42.022	**31(12)** 1928(10)	7-SAT r=86.674	**3499(8)** 4340(7)
4-SAT r=9.627	**133**(12) 192(12)	5-SAT r=20.756	**3468(8)** 4452(7)	6-SAT r=42.359	1003(11) **227(12)**	7-SAT r=86.953	3415(8) **2646(9)**
4-SAT r=9.729	243(12) **212**(12)	5-SAT r=20.876	5118(6) **3051(8)**	6-SAT r=42.696	**2757(9)** 3528(8)	7-SAT r=87.232	150(12) **135**(12)
4-SAT r=9.83	1252(11) **344(12)**	5-SAT r=20.997	5100(6) **4436(7)**	6-SAT r=43.033	4418(7) **3647(8)**	7-SAT r=87.511	2589(9) **1054(11)**
4-SAT r=9.931	149(12) **118**(12)	5-SAT r=21.117	2522(9) **1772(10)**	6-SAT r=43.37	939(11) **165(12)**	7-SAT r=87.79	899(11) **74(12)**

Table 15: **Comparing CScoreSAT and HScoreSAT at each ratio of random k-SAT ($k > 3$) in SAT Challenge 2012 benchmark.** Each solver is executed one time on each instance, with a cutoff time of 1000 seconds. Each cell reports the result of CScoreSAT in the upper row and that of HScoreSAT in the lower row, in the form of "par10(#solved)". We color the ratios gray at which HScoreSAT performs better than CScoreSAT. Note that for 6-SAT at $r = 42.696$ where the results seem a little odd, we conduct an additional experiment executing both solvers 10 times on each instance, and HScoreSAT succeeds in 84 runs while CScoreSAT succeeds in 83 runs.

	4-SAT	5-SAT	6-SAT	7-SAT
(r^*_{min}, r^*_{max})	(9.627,9.729)	(20.756,20.876)	(42.022,42.359)	(86.674,86.953)

Table 16: **The conjectured interval of the boundary ratio r^*.** HScoreSAT has worse performance than CScoreSAT at ratios $r \leq r^*_{min}$, and has better (or at least competitive) performance at ratios $r \geq r^*_{max}$, based on experiments on the SAT Challenge 2012 benchmark.

5. On Computation of $score_2$

For algorithms employing scoring functions based on $score_2$, such as CScoreSAT and HScoreSAT, the computation of $score_2$ has a considerable impact on their efficiency. In this section, we investigate the computation issues of $score_2$. Particularly, we propose a cache-based implementation and analyze its time complexity in each flip. We also measure its overhead in the two algorithms through experiments.

5.1 Implementation and Complexity of Computing $score_2$

We propose a caching implementation for computing variables' $score_2$ values, which is inspired by the caching implementation for computing variable scores. Typically, not all variable scores (or $score_2$ values) change after each search step; this suggests that rather than recomputing all variable scores (or $score_2$ values) in each step, it should be more efficient to compute all scores (or $score_2$ values) when the search is initialized, but to subsequently only update the scores (or $score_2$ values) affected by a variable that has been flipped (Hoos & Stützle, 2004).

In our caching implementation, the $score_2$ values of all variables are computed when the search is initialized, and are subsequently updated in each flip. The initialized computation of $score_2$ is straightforward according to the definition of $score_2$ and will not be discussed. Comparatively, the procedure of updating $score_2$ values are of much more interest.

To facilitate describing the procedure of updating $score_2$ values and analyzing its time complexity, we first introduce some notations and definitions below.

Given a CNF formula F,

- for a variable $x \in V(F)$, $CL(x) = \{c | c$ is a clause in F and x appears in $c\}$;
- for a clause $c \in F$, c.num_true_lit is the number of true literals in c;
- for a clause $c \in F$ with exactly two true literals, we use true_lit_var(c) and true_lit_var2(c) to record the two corresponding variables of the two true literals in c (these two notations will only be used in pseudo-code for the sake of formalization);
- we use x^* to denote the variable flipped in the current step;
- n, m, k, r is the number of variables, the number of clauses, the maximum clause length, and the clause-to-variable ratio.

Definition 5. *Given a clause c and a variable x, we say the contribution of clause c to $score_2(x)$ is +1 if flipping x would cause c transform from 1-satisfied to 2-satisfied, -1 if flipping x would cause c transform from 2-satisfied to 1-satisfied, and 0 otherwise.*

A useful observation is that for a variable $x \in V(F)$, $score_2(x)$ equals the sum of contributions of all clauses to it. Also, it is obvious that clauses in which x does not appear always contribute 0 to $score_2(x)$.

Now we describe in detail the procedure of updating $score_2$ values in each flip, whose pseudo-code is shown in Algorithm 3. After flipping x^*, according to the definition of $score_2$, $score_2(x^*)$ just changes to its opposite number (lines 1 and 21). The essential part is updating $score_2$ values for variables that share clauses with x^* (since other variables would not change their $score_2$ values), which is accomplished with a loop (lines 2-20). In each iteration of the loop, a clause $c \in CL(x^*)$ is considered and necessary updates are performed, according to two different cases: either the literal of x^* in c becomes a true literal, or not. Here we explain the updates for the first case, and those for the other case can be understood similarly.

In the first case (i.e., the literal of x^* in c becomes a true literal), along with the flip of x^*, c.num_true_lit is increased by 1. Suppose c.num_true_lit changes from $t-1$ to t. If neither $t-1$ nor t is 1 or 2, then flipping x^* causes no change to any variable's $score_2$ (easy to see from the definition of $score_2$). So, we only need to consider the following three cases.

Algorithm 3: updating $score_2$ values in a flip step

```
1  org_score2(x*) := score2(x*);
2  for each c ∈ CL(x*) do
3      if the literal of x* in c becomes a true literal then
4          c.num_true_lit+=1;
5          if c.num_true_lit=3 then
6              score2(true_lit_var(c))+=1;
7              score2(true_lit_var2(c))+=1;
8          else if c.num_true_lit=2 then
9              for each x_i ∈ c do score2(x_i)−=1;
10         else if c.num_true_lit=1 then
11             for each x_i ∈ c do score2(x_i)+=1;
12     else
13         c.num_true_lit−=1;
14         if c.num_true_lit=2 then
15             score2(true_lit_var(c))−=1;
16             score2(true_lit_var2(c))−=1;
17         else if c.num_true_lit=1 then
18             for each x_i ∈ c do score2(x_i)+=1;
19         else if c.num_true_lit=0 then
20             for each x_i ∈ c do score2(x_i)−=1;
21     score2(x*) := −org_score2(x*);
```

- c.num_true_lit changes from 2 to 3: Before flipping x^*, c had two true literals, and let us denote their corresponding variables as y_1 and y_2. The contributions of c to $score_2(y_1)$ and $score_2(y_2)$ were both -1 before flipping x^*. After flipping x^*, c becomes a 3-satisfied clause, and the contributions of c to $score_2(y_1)$ and $score_2(y_2)$ both become 0. Hence, along with flipping x^*, the changes on $score_2(y_1)$ and $score_2(y_2)$ are both 0-(-1)=+1 (lines 6-7). For other variables in c, either before or after the flip of x^*, the contributions of c to their $score_2$ values are 0.

- c.num_true_lit changes from 1 to 2: Before flipping x^*, c had only one true literal, and let us denote its corresponding variable as y_1. The contribution of c to $score_2(y_1)$ was 0 before flipping x^*, but becomes -1 after flipping x^*, indicating a change of -1 on $score_2(y_1)$. For other variables in c (except x^*), the contributions of c to their $score_2$ values were +1 before flipping x^*, but become 0 after flipping x^*, indicating a change of -1 on their $score_2$ values. Therefore, for all variables in c (except x^*), along with flipping x^*, the changes on their $score_2$ values are -1 (lines 8-9).

  Note that we include x^* in the loop (line 9) just in order to save computational consumption. As we have a special update for $score_2(x^*)$, any change on $score_2(x^*)$ between line 1 and line 21 has no impact in effect.

- c.num_true_lit changes from 0 to 1: Before flipping x^*, for all variables in c, the contributions of c to their $score_2$ values were 0. After flipping x^*, c becomes 1-satisfied (the true literal's corresponding variable is x^*), and for all variables in c (except x^*), the contributions of c to their $score_2$ values become +1. Therefore, for all variables in c (except x^*), along with flipping x^*, the changes on their $score_2$ values are +1 (lines 10-11).

The complexity of the $score_2$ updating procedure in each flip is determined by the main loop (lines 2-20). When flipping a variable x, there are $|CL(x)|$ items in the main loop, and in each iteration there are three possible cases, where the first case requires only 2 operations[5] and the latter two require $\Theta(k)$ operations. Therefore, the worst-case time complexity of the $score_2$ updating procedure in each flip is $\Theta(max_{x \in V(F)}|CL(x)| \cdot max\{2, k, k\})$=$\Theta(max_{x \in V(F)}|CL(x)| \cdot k)$.

For uniform random k-SAT formulas with clause-to-variable ratio r, there are totally $m \cdot k$ literals and each variable is expected to have $mk/n = kr$ literals. That is, each variable $x \in V(F)$ is expected to appear in kr clauses (when n approaches to $+\infty$, this is true with probability almost 1), i.e., $|CL(x)| \approx kr$. Therefore, the complexity of the $score_2$ updating procedure in each flip becomes $\Theta(kr \cdot k) = \Theta(k^2 r)$.

Fortunately, for uniform random k-SAT formulas with constant ratio r, both k and r are constants. Therefore, independent of instance size, this implementation of $score_2$ computation achieves a time complexity of $\Theta(1)$ for each search step, just as the caching implementation of $score$ computation does (referring to pages 272-273 in (Hoos & Stützle, 2004)).

5.2 Computational Overhead of Computing $score_2$

In this subsection, we study the computational overhead of computing $score_2$ in CScoreSAT and HScoreSAT. We carry out experiments with the Threshold Comp13 benchmark to figure out the CPU time per 10^7 steps for computing $score_2$ and its percentage in the total CPU time of the solver per 10^7 steps.

	4-SAT	5-SAT	6-SAT	7-SAT
CScoreSAT	13.9	28.5	52.4	94.6
computing $score_2$	1.6	8.0	17.9	37.1
percentage	11.5%	28.1%	34.2%	39.2%
HScoreSAT	14.3	28.7	53.2	95.1
computing $score_2$	1.6	8.4	17.8	36.9
percentage	11.2%	29.3%	33.5%	38.8%

Table 17: CPU time consumption (in seconds) per 10^7 steps for CScoreSAT and HscoreSAT, and for computing $score_2$, as well as their ratios. The results are averaged over all instances in the Threshold Comp13 benchmark.

Our investigation shows that the overhead of computing $score_2$ occupies a considerable percentage of the solvers' whole run time, ranging from 11% to 40%. Nevertheless, since $score_2$ is critical to the solvers, this price indeed pays off. Further observation reveals that more than 90% of

5. note that true_lit_var(c) and true_lit_var2(c) are recorded initially for accelerating updating variable scores, and thus we do not need extra price to maintain them for $score_2$ updates.

the solvers' run time are due to the flip function, where two most costly parts are $score$ and $score_2$ updates. Another interesting phenomenon is that the percentage of overhead caused by $score_2$ computation rises as the clause length increases, although $score_2$ is less used (for tie-breaking) for longer clauses (see Table 8). The reason might be that as the clause length increases, the portion of variables whose $score_2$ values need to be updated is increasing compared to that of variables whose $scores$ need to be updated.

6. Summary and Future Work

In this paper, we proposed three new scoring functions based on $score_2$ for improving SLS algorithms on random SAT instances with long clauses. Despite their simplicity, the proposed scoring functions are very effective, and the resulting SLS algorithms namely CScoreSAT and HscoreSAT show excellent performance on random k-SAT instances with long clauses.

First, we combined the $score$ and $score_2$ properties to design a scoring function named $cscore$ (comprehensive score), which aims to improve the greedy mode by combining greediness and look-ahead greediness. We also defined comprehensively decreasing variables accordingly. We further proposed the $hscore$ function combining $cscore$ with the diversification age, which is devoted to improving the diversification mode. These two scoring functions were used to develop the CScoreSAT algorithm. The experiments show that the performance of CScoreSAT exceeds that of state-of-the-art SLS solvers by orders of magnitudes on large random 5-SAT and 7-SAT instances near phase transition. Moreover, CScoreSAT significantly outperforms its competitors on random k-SAT instances with various ratios for each $k \in \{4, 5, 6, 7\}$ from SAT Challenge 2012.

To improve CScoreSAT for solving random k-SAT instances at the threshold ratio of phase transition, we propose another scoring function called $hscore_2$, which combines $score_2$ and age. By using $hscore_2$ to break ties and adjust the greedy mode accordingly, we obtain the HScoreSAT algorithm. Experiments on random k-SAT instances at phase-transition threshold show that HScoreSAT significantly improves CScoreSAT and outperforms state-of-the-art SLS algorithms.

Finally, as the $score_2$ property is a key notion in our algorithms, we also present the implementation details of $score_2$ computation, and analyze its complexity per flip and its computational overhead.

As for future work, a significant research issue is to improve SLS algorithms for structured instances by $score_2$-based scoring functions. Furthermore, the notions in this work are so simple that they can be easily applied to other problems, such as constrained satisfaction and graph search problems.

Acknowledgement

This work is supported by China National 973 Program 2010CB328103 and 2014CB340301, National Natural Science Foundation of China 61370072 and 61472369, and ARC Grant FT0991785. We would like to thank the anonymous referees for their helpful comments on the earlier versions of this paper, which significantly improve the quality of this paper.

References

Abramé, A., Habet, D., & Toumi, D. (2014). Improving configuration checking for satisfiable random k-SAT instances. In *Proc. of ISAIM-14*.

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

Balint, A., Biere, A., Fröhlich, A., & Schöning, U. (2014). Improving implementation of SLS solvers for SAT and new heuristics for k-sat with long clauses. In *Proc. of SAT-14*, pp. 302–316.

Balint, A., & Fröhlich, A. (2010). Improving stochastic local search for SAT with a new probability distribution. In *Proc. of SAT-10*, pp. 10–15.

Balint, A., & Schöning, U. (2012). Choosing probability distributions for stochastic local search and the role of make versus break. In *Proc. of SAT-12*, pp. 16–29.

Braunstein, A., Mézard, M., & Zecchina, R. (2005). Survey propagation: An algorithm for satisfiability. *Random Struct. Algorithms*, *27*(2), 201–226.

Cai, S., & Su, K. (2012). Configuration checking with aspiration in local search for SAT. In *Proc. of AAAI-12*, pp. 334–340.

Cai, S., & Su, K. (2013a). Comprehensive score: Towards efficient local search for SAT with long clauses. In *Proc. of IJCAI-13*, pp. 489–495.

Cai, S., & Su, K. (2013b). Local search for Boolean Satisfiability with configuration checking and subscore. *Artif. Intell.*, *204*, 75–98.

Cai, S., Su, K., & Luo, C. (2013a). Improving walksat for random k-satisfiability problem with k > 3. In *Proc. of AAAI-13*, pp. 145–151.

Cai, S., Su, K., Luo, C., & Sattar, A. (2013b). NuMVC: An efficient local search algorithm for minimum vertex cover. *J. Artif. Intell. Res. (JAIR)*, *46*, 687–716.

Cai, S., Su, K., & Sattar, A. (2011). Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artif. Intell.*, *175*(9-10), 1672–1696.

Chieu, H. L., & Lee, W. S. (2009). Relaxed survey propagation for the weighted maximum satisfiability problem. *J. Artif. Intell. Res. (JAIR)*, *36*, 229–266.

Gent, I. P., & Walsh, T. (1993). Towards an understanding of hill-climbing procedures for SAT. In *Proc. of AAAI-93*, pp. 28–33.

Hoos, H. H., & Stützle, T. (2004). *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., & Leyton-Brown, K. (2009). Satenstein: Automatically building local search SAT solvers from components. In *Proc. of IJCAI-09*, pp. 517–524.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*, 671–680.

Kirkpatrick, S., & Selman, B. (1994). Critical behavior in the satisfiability of random boolean formulae. *Science*, *264*, 1297–1301.

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.

Li, C. M., Huang, C., & Xu, R. (2014). Balance between intensification and diversification: a unity of opposites. In *Proc. of SAT Competition 2014: Solver and Benchmark Descriptions*, pp. 10–11.

Li, C. M., & Huang, W. Q. (2005). Diversification and determinism in local search for satisfiability. In *Proc. of SAT-05*, pp. 158–172.

Li, C. M., & Li, Y. (2012). Satisfying versus falsifying in local search for satisfiability - (poster presentation). In *Proc. of SAT-12*, pp. 477–478.

Luo, C., Cai, S., Wu, W., Jie, Z., & Su, K. (2014). CCLS: An efficient local search algorithm for weighted maximum satisfiability. *IEEE Transactions on Computers*, in press.

Luo, C., Cai, S., Wu, W., & Su, K. (2013). Focused random walk with configuration checking and break minimum for satisfiability. In *Proc. of CP-13*, pp. 481–496.

Luo, C., Cai, S., Wu, W., & Su, K. (2014). Double configuration checking in stochastic local search for satisfiability. In *Proc. of AAAI-14*, pp. 2703–2709.

Luo, C., Su, K., & Cai, S. (2012). Improving local search for random 3-SAT using quantitative configuration checking. In *Proc. of ECAI-2012*, pp. 570–575.

Mertens, S., Mézard, M., & Zecchina, R. (2006). Threshold values of random k-SAT from the cavity method. *Random Struct. Algorithms*, *28*(3), 340–373.

Mézard, M. (2003). Passing messages between disciplines. *Science*, *301*, 1685–1686.

Pham, D. N., Thornton, J., Gretton, C., & Sattar, A. (2007). Advances in local search for satisfiability. In *Australian Conference on Artificial Intelligence*, pp. 213–222.

Prestwich, S. D. (2005). Random walk with continuously smoothed variable weights. In *Proc. of SAT-05*, pp. 203–215.

Selman, B., Kautz, H. A., & Cohen, B. (1994). Noise strategies for improving local search. In *Proc. of AAAI-94*, pp. 337–343.

Thornton, J., Pham, D. N., Bain, S., & Ferreira Jr., V. (2004). Additive versus multiplicative clause weighting for SAT. In *Proc. of AAAI-04*, pp. 191–196.

Tompkins, D. A. D., Balint, A., & Hoos, H. H. (2011). Captain jack: New variable selection heuristics in local search for SAT. In *Proc. of SAT-11*, pp. 302–316.

Tompkins, D. A. D., & Hoos, H. H. (2010). Dynamic scoring functions with variable expressions: New SLS methods for solving SAT. In *Proc. of SAT-10*, pp. 278–292.

Xu, L., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2008). SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res. (JAIR)*, *32*, 565–606.