Comprehensive Score: Towards Efficient Local Search for SAT with Long Clauses

Shaowei Cai and **Kaile Su**
Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Australia
Key Laboratory of High Confidence Software Technologies, Peking University, Beijing, China
shaoweicai.cs@gmail.com; k.su@griffith.edu.au

Abstract

It is widely acknowledged that stochastic local search (SLS) algorithms can efficiently find models of satisfiable formulae for the Boolean Satisfiability (SAT) problem. There has been much interest in studying SLS algorithms on random k-SAT instances. Compared to random 3-SAT instances which have special statistical properties rendering them easy to solve, random k-SAT instances with long clauses are similar to structured ones and remain very difficult. This paper is devoted to efficient SLS algorithms for random k-SAT instances with long clauses. By combining a novel variable property $subscore$ with the commonly used property $score$, we design a scoring function named *comprehensive score*, which is utilized to develop a new SLS algorithm called CScoreSAT. The experiments show that CScoreSAT outperforms state-of-the-art SLS solvers, including the winners of recent SAT competitions, by one to two orders of magnitudes on large random 5-SAT and 7-SAT instances. In addition, CScoreSAT significantly outperforms its competitors on random k-SAT instances for each $k = 4, 5, 6, 7$ from SAT Challenge 2012, which indicates its robustness.

1 Introduction

The Boolean Satisfiability (SAT) problem is a prototypical NP-complete problem whose task is to decide for a given Boolean formula whether it has a model. 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 [Kautz *et al.*, 2009].

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 that they cannot prove an instance to be unsatisfiable, they often find models of satisfiable formulae surprisingly effectively [Hoos and Stützle, 2004].

SLS algorithms are often evaluated on random k-SAT benchmarks [Achlioptas, 2009]. 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 (further denoted by r), they provide adjustable hardness levels to assess the solving capabilities. Moreover, heuristics for SLS algorithms to solve random k-SAT instances may be potentially beneficial for solving realistic problems.

However, random 3-SAT instances exhibit some particular statistical properties which render them easy to solve, for example, by SLS algorithms and a statistical physics approach called Survey Propagation [Braunstein *et al.*, 2005]. Recent studies show that SLS algorithms such as WalkSAT [Selman *et al.*, 1994] and probSAT [Balint and Schöning, 2012], can solve random 3-SAT instances near the phase transition with half million variables very quickly [Kroc *et al.*, 2010; Balint and Schöning, 2012].

In contrast, random k-SAT instances with long clauses are, to some extend, similar to structured ones [Balint and Schöning, 2012], and remain very difficult. Although the very recent solver CCASat [Cai and Su, 2013] made a breakthrough in solving such instances, the performance of SLS algorithms on this kind of benchmarks lags far behind that on 3-SAT ones. On the other hand, it is believed that improving SLS algorithms for random instances with large clause length would yield improvements for non-random ones [Balint and Schöning, 2012]. Therefore, it is of great significance to improve SLS algorithms for these instances.

The aim of this paper is thus to design a more efficient SLS algorithm for solving random k-SAT instances with long clauses. We proposed a variable property called $subscore$ [Cai and Su, 2013], which shares the same spirit with the commonly used property $score$ [Selman *et al.*, 1992]. While $score$ measures the increment of satisfied clauses by flipping a variable, $subscore$ does that of clauses with more than one true literal. The notion of $subscore$ considers transformations among satisfied clauses, distinguishing itself from previous variable properties. In this work, we design a scoring function called *comprehensive score*, which is a linear combination of $score$ and $subscore$. We also define a new type of "decreasing" variables namely *comprehensively decreasing* variables.

Based on the notions of comprehensive score and comprehensively decreasing variable, we develop a new SLS algorithm called CScoreSAT (comprehensive score based SAT algorithm). We conduct extensive experiments to compare CScoreSAT against state-of-the-art SLS solvers including Sparrow2011, sattime2012, EagleUP, probSAT and CCASat. The experiments on large random 5-SAT and 7-SAT instances show that CScoreSAT outperforms its competitors by one to two orders of magnitudes, in terms of success rate or run time. In particular, CScoreSAT achieves a success rate of 62% for random 5-SAT instances with 4500 variables and 11% for 7-SAT ones with 300 variables, whereas all its competitors fail to find a satisfying assignment for any of these instances. Moreover, CScoreSAT significantly outperforms its competitors on k-SAT instances for each $k = 4, 5, 6, 7$ with various ratios from SAT Challenge 2012, indicating its robustness.

The rest of this paper is organized as follows. Some definitions and notations are given in the next section. Section 3 presents the $subscore$ notion, while Section 4 presents the comprehensive score function and the concept of comprehensively decreasing variable. The CScoreSAT algorithm is described in Section 5, and experimental results demonstrating the performance of CScoreSAT are presented in Section 6. Finally, we summarize the main contributions in this work and outline directions for future research.

2 Preliminaries

Given a set of n Boolean $variables$ $\{x_1, x_2, ..., x_n\}$, a $literal$ is either a variable x or its negation $\neg x$, 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 (SAT) 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.

For a CNF formula F, we use $V(F)$ to denote the set of all variables that appear in F. Two variables are neighbors if and only if there is at least one clause in which both variables appear. The neighbourhood of a variable x is $N(x) = \{y|y$ occurs in at least one clause with $x\}$. Given an assignment α, if a literal evaluates to true, we say it is a $true\ literal$; otherwise, we say it is a $false\ literal$. A clause is $satisfied$ if it has at least one true literal, and $unsatisfied$ 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 is $score$, which measures the increase in the number of satisfied clauses by flipping a variable. In dynamic local search (DLS) algorithms which use clause weighting techniques, $score$ measures the increase in the total weight of satisfied clauses by flipping a variable. The age of a variable is defined as the number of search steps that have occurred since the variable was last flipped. A variable is $decreasing$ if its $score$ is positive, and $increasing$ if its $score$ is negative. A variable is $configuration\ changed$ if and only if since its last flip, at least one of its neighbors has been flipped [Cai and Su, 2012].

3 Subscore

In this section, we introduce a novel variable property called $subscore$ [Cai and Su, 2013], which is an important concept in this work.

From the perspective of global optimization, the goal of the SAT problem is to minimize the number of unsatisfied clauses. If this minimum value equals 0, then the instance is satisfiable. Thus, it is natural for SLS algorithms to select a variable to flip according to the $score$ property, which concerns the transformations between satisfied and unsatisfied clauses if the variable were to be flipped.

However, simply categorizing clauses into satisfied ones and unsatisfied ones is not informative enough to guide the local search, especially for instances with long clauses. We may consider the number of true literals in a clause, which can be regarded as the degree of being satisfied of the clause. This is quite intuitive, as the more true literals a clause contains, the less likely it would become unsatisfied in the following flips.

In the following, we propose a definition which captures the degree of a clause being satisfied.

Definition 1. Given a CNF formula F and an assignment α to its variables, the *satisfaction degree* of a clause C, is defined as the number of true literals in C under α. A clause with a satisfaction degree of δ is said to be a δ-satisfied clause.

In this work, when we are talking about δ-satisfied clauses, the assignment α is usually explicit from the context, and thus omitted.

According to the above definition, a δ-satisfied clause is satisfied if $\delta > 0$, and unsatisfied otherwise. Among satisfied clauses, 1-satisfied clauses are the most unstable, as they can become unsatisfied by flipping only one variable. Thus, it is beneficial for SLS algorithms to take into account the transformations between 1-satisfied and 2-satisfied clauses.

Based on the above considerations, the variable property $subscore$ is defined as follows.

Definition 2. For a variable x, $subscore(x)$ is defined as $submake(x)$ minus $subbreak(x)$, where $submake(x)$ is the number of 1-satisfied clauses that would become 2-satisfied by flipping x, and $subbreak(x)$ is the number of 2-satisfied clauses that would become 1-satisfied by flipping x.

When considering clause weights in DLS algorithms, $submake(x)$ measures the total weight of the 1-satisfied clauses that would become 2-satisfied by flipping x, and $subbreak(x)$ does that of the 2-satisfied clauses that would become 1-satisfied by flipping x.

Although we also tried similar property that measures transformations between 2-satisfied and 3-satisfied clauses, we did not find it useful in our algorithm.

4 Comprehensive Score and Comprehensively Decreasing Variables

In this section, we design the scoring function comprehensive score, which is a linear combination of the $score$ and $subscore$ properties. Based on this function, we also define the concept of comprehensively decreasing variable.

4.1 Comprehensive Score

The $score$ property characterizes the greediness of flipping a variable at the current search step, as it tends to decrease the number of unsatisfied clauses, which is indeed the aim of the SAT problem. On the other hand, the $subscore$ 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.

To combine the greediness and look-ahead greediness together, we consider designing a scoring function that incorporates both $score$ and $subscore$. When deciding the candidate variables' priorities of being selected, although $score$ is more important than $subscore$, in some cases $subscore$ should be allowed to overwrite the priorities. For example, for two variables which have a relatively small $score$ difference and a significant $subscore$ difference, it is advisable to prefer to flip the one with greater $subscore$.

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

- First, the $score$ property plays a more important role;
- Second, the $subscore$ 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 3. For a CNF formula F, the *comprehensive score* function, denoted by $cscore$, is a function on $V(F)$ such that

$$cscore(x) = score(x) + \lfloor subscore(x)/d \rfloor,$$

where d is a positive integer parameter.

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

4.2 Comprehensively Decreasing Variables

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

While the condition that $cscore(x) > 0$ 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 *et al.*, 1983]), unless the algorithm gets stuck in a local optimum.

Comprehensively decreasing variables are considered to be the flip candidates in the greedy search phases of our algorithm. To avoid blind search, we utilize the configuration checking (CC) strategy [Cai and Su, 2012] to identify the "good" comprehensively decreasing variables which are *configuration changed*. For convenience, such variables are further referred to as CDCC (Comprehensively Decreasing and Configuration Changed) variables. Note that the CC strategy was proposed to handle the revisiting problem in local search [Cai *et al.*, 2011], and has proved effective in SLS algorithms for SAT [Cai and Su, 2012; Luo *et al.*, 2012].

5 The CScoreSAT Algorithm

This section presents the CScoreSAT algorithm, which utilizes two key notions: comprehensive score and comprehensively decreasing variable.

For the sake of diversification, CScoreSAT also employs the PAWS clause weighting scheme [Thornton *et al.*, 2004]. All clause weights are initiated as 1. When a local optimum is reached, 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 unsatisfied clauses are increased by one.

For convenience, before getting into the details of the CScoreSAT algorithm, we first introduce the two **scoring functions used in CScoreSAT**.

1. For the greedy search, CScoreSAT adopts the $cscore$ function.
2. When reaching a local optimum, CScoreSAT makes use of a hybrid scoring function (denoted by $hscore$), which combines $cscore$ with the diversification property age:

$$hscore(x) = cscore(x) + \lfloor age(x)/\beta \rfloor,$$

where β is a (relatively large) positive integer parameter.

Algorithm 1: CScoreSAT

```
   Input: CNF-formula $F$, $maxSteps$
   Output: A satisfying assignment $\alpha$ of $F$, or "unknown"
1  begin
2      $\alpha$ := randomly generated truth assignment;
3      for $step := 1$ to $maxSteps$ do
4          if $\alpha$ satisfies $F$ then return $\alpha$;
5          if $\exists$ CDCC variables then
6              $v$ := the CDCC variable with the greatest $cscore$,
               breaking ties in favor of the oldest one;
7          else
8              update clause weights according to PAWS;
9              pick a random unsatisfied clause $C$;
10             $v$ := the variable in $C$ with the greatest $hscore$,
               breaking ties in favor of the oldest one;
11         $\alpha := \alpha$ with $v$ flipped;
12     return "unknown";
13 end
```

The CScoreSAT algorithm is outlined in Algorithm 1, as described below. 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 works in two modes. In each search step, it works in either the greedy mode or the diversification mode, depending on the existence of CDCC variables.

If there exist CDCC variables, CScoreSAT works in the greedy mode. It picks the CDCC variable with the greatest $cscore$ value to flip, breaking ties by preferring the oldest one.

If no CDCC variable is present, which means a local optimum is identified, then CScoreSAT switches to the diversification mode. It first updates clause weights according to the PAWS scheme. Then it randomly selects an unsatisfied clause C, and picks the variable from C with the greatest $hscore$ value to flip, breaking ties by favoring the oldest one.

We conclude this section by some comments on the parameters of CScoreSAT. CScoreSAT has three parameters, namely sp, d and β. The sp parameter is for PAWS, and the latter two are introduced by our scoring functions. Fortunately, β is a constant (2000) for any instance, and d is simply defined as $13 - k$ in this work. As for sp, it becomes insensitive in our algorithm. Actually, it is set to the same value for all 4-SAT and 5-SAT instances, and is set to another fixed value for all 6-SAT and 7-SAT instances.

6 Experimental Evaluations

We carry out extensive experiments to evaluate CScoreSAT on random k-SAT instances with $k > 3$. For each benchmark, we compare CScoreSAT with state-of-the-art SLS solvers.

6.1 The Benchmarks

All the instances used in our experiments are generated according to the random k-SAT model at or near the solubility phase transition, which are the most difficult among random k-SAT instances [Kirkpatrick and Selman, 1994]. Specifically, we adopt the following five benchmarks.

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.

6.2 Experimental Setup

CScoreSAT is implemented in C++ and complied by g++ with the '-O2' option. The sp parameter for the PAWS scheme is set to 0.62 for k-SAT with $3 < k \leq 5$, and 0.9 for k-SAT with $k > 5$. The d and β parameters in the $cscore$ and $hscore$ functions are set as $d = 13 - k$ (k is the clause length of the k-SAT formula) and $\beta = 2000$. To run CScoreSAT, we only need to specify the sp parameter, as d is computed from the input instance and β is a fixed constant for all instances.

We compare CScoreSAT with five state-of-the-art SLS solvers, including **Sparrow2011** [Balint and Fröhlich, 2010], **sattime2012** [Li and Li, 2012], **EagleUP** [Gableske and Heule, 2011], **CCASat** [Cai and Su, 2013], and **probSAT** (the adaptive version) [Balint and Schöning, 2012]. The first three are the best SLS solvers (or their improved versions) from SAT Competition 2011, and CCASat is the winner of the random track of SAT Challenge 2012, while probSAT is one of the best SLS solvers in the recent literature. For Sparrow2011 and EagleUP, we use the binaries from SAT Competition 2011[1]. The binary of CCASat can be downloaded online[2], while the binaries of sattime2012 and probSAT are kindly provided by their authors.

All experiments are carried out on a workstation under Linux, using 2 cores of Intel(R) Core(TM) i7-2640M 2.8 GHz CPU and 8 GB RAM. The experiments are conducted with EDACC [Balint *et al.*, 2010], an experimental platform for testing SAT solvers, which has been used for SAT Challenge 2012. 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 fourth benchmarks, and 1000 seconds for the SAT Challenge 2012 benchmark (close to the cutoff in SAT Challenge 2012, $i.e.$, 900 seconds).

6.3 Evaluation Methodology

For the instances from SAT Competition 2011, we run each solver 10 times for each instance and thus 100 runs for each size. For the instances randomly generated (100 instances for each instance class) and the SAT Challenge 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"), as well as the par10 run time ("par10 time"), which is a penalized average run time where a timeout of a solver is penalized as 10$*$(cutoff time). Note that the par10 run time is adopted in SAT competitions and has been widely used in the literature as a prominent performance measure for SLS-based SAT solvers [KhudaBukhsh *et al.*, 2009; Tompkins and Hoos, 2010; Tompkins *et al.*, 2011; Balint and Schöning, 2012]. The results in **bold** indicate the best performance for an instance class.

[1]http://www.cril.univ-artois.fr/SAT11/solvers/SAT2011-static-binaries.tar.gz

[2]http://www.shaoweicai.net/Code/Binary-of-CCASat.tar.gz

	5sat750	5sat1000	5sat1250	5sat1500	5sat2000
	suc runs	suc runs	suc runs	suc runs	suc runs
	par10 time	par10 time	par10 time	par10 time	par10 time
CScoreSAT	100	100	100	100	**100**
	35	**38**	**47**	**145**	**289**
CCASat	100	100	100	100	93
	47	81	128	443	4386
probSAT	100	100	100	98	71
	88	185	237	1753	15635
Sparrow2011	100	100	100	99	72
	51	159	174	1231	15288
sattime2012	100	100	95	56	14
	754	1254	5288	24101	43249
EagleUP	100	100	100	88	25
	72	184	384	7223	38031

Table 1: **Experimental results on the 5-SAT Comp11 benchmark.** The results are based on 100 runs for each solver on each instance class, with a cutoff time of 5000 seconds.

	5sat3000	5sat3500	5sat4000	5sat4500	5sat5000
	suc runs	suc runs	suc runs	suc runs	suc runs
	par10 time	par10 time	par10 time	par10 time	par10 time
CScoreSAT	**100**	**100**	**87**	**62**	**38**
	694	**1431**	**8167**	**21513**	**32005**
CCASat	64	35	10	0	0
	19403	33540	45287	n/a	n/a
probSAT	40	6	3	0	0
	30867	47188	48591	n/a	n/a
Sparrow2011	31	8	4	0	0
	35360	46147	48080	n/a	n/a
sattime2012	0	0	0	0	0
	n/a	n/a	n/a	n/a	n/a
EagleUP	1	0	0	0	0
	49540	n/a	n/a	n/a	n/a

Table 2: **Experimental results on the 5-SAT Huge benchmark.** The results are based on 100 runs for each solver on each instance class, with a cutoff time of 5000 seconds.

	7sat150	7sat200	7sat250	7sat300	7sat400
	suc runs	suc runs	suc runs	suc runs	suc runs
	par10 time	par10 time	par10 time	par10 time	par10 time
CScoreSAT	100	**90**	**35**	**11**	0
	131	**5853**	**34070**	**44776**	n/a
CCASat	100	72	7	0	0
	232	14912	46731	n/a	n/a
probSAT	88	11	0	0	0
	6980	44806	n/a	n/a	n/a
Sparrow2011	100	17	0	0	0
	642	41912	n/a	n/a	n/a
sattime2012	100	49	2	0	0
	498	26998	49095	n/a	n/a
EagleUP	98	48	2	0	0
	1345	27052	49076	n/a	n/a

Table 3: **Experimental results on the 7-SAT Comp11 benchmark.** The results are based on 100 runs for each solver on each instance class, with a cutoff time of 5000 seconds.

	7sat220	7sat240	7sat260	7sat280	7sat300
	suc runs	suc runs	suc runs	suc runs	suc runs
	par10 time	par10 time	par10 time	par10 time	par10 time
CScoreSAT	**83**	**66**	**53**	**24**	**11**
	10639	**17901**	**24825**	**39283**	**44889**
CCASat	68	33	9	5	0
	17189	34158	45736	47605	n/a
probSAT	10	2	0	0	0
	45253	49052	n/a	n/a	n/a
Sparrow2011	13	2	0	0	0
	43407	49051	n/a	n/a	n/a
sattime2012	39	13	4	0	0
	31868	43935	48113	n/a	n/a
EagleUP	34	11	3	0	0
	33762	44832	48635	n/a	n/a

Table 4: **Experimental results on the 7-SAT Random benchmark.** The results are based on 100 runs for each solver on each instance class, with a cutoff time of 5000 seconds.

6.4 Experimental Results

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

Results on 5-SAT Comp11 Benchmark:

Table 1 shows the comparative results on the 5-SAT Comp11 benchmark. As is clear from Table 1, CScoreSAT shows significant superiority 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 5sat2000 instances, which are of the largest size in SAT competitions, the par10 run time 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 2. 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 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 5sat3500 instances. Furthermore, CScoreSAT succeeds in 62 and 38 runs for the 5sat4500 and 5sat5000 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.

Results on 7-SAT Comp11 Benchmark:

Table 3 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 (among which CCASat has the highest success rate of 7%) on the 7sat250 instances, while CScoreSAT still achieves a success rate of 35% for this instance class.

	4sat #solved par10 time	5sat #solved par10 time	6sat #solved par10 time	7sat #solved par10 time	over all #solved par10 time
CScoreSAT	**119** **174**	**84** **3146**	**110** **935**	**91** **2559**	**404** **1703**
CCASat	112 751	71 4264	99 1887	77 3734	359 2659
probSAT	111 778	54 5657	76 3877	57 5380	298 3923
Sparrow2011	79 3514	52 5812	72 4193	65 4714	268 4558
EagleUP	40 6784	42 6615	93 2466	76 3806	251 4917
sattime2012	49 6031	32 7407	84 3187	81 3422	246 5011

Table 5: **Experimental results on SAT Challenge 2012 benchmark.** Each instance class contains 120 instances, and each solver is performed once on each instance with a cutoff time of 1000 seconds.

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. As reported in Table 4, 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 7sat220 to 7sat260, the success rates of all the competitors decline eight times or more, whereas that of CScoreSAT drops only thirty percents. When coming to the 7sat260 instances, probSAT and Sparrow2011 fail in all runs, and the other three 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 5 reports the number of solved instances and the par10 time 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 on the run time distribution of CScoreSAT show that CScoreSAT solves 365 instances within one half cutoff time, whereas none of its competitors solves more than 360 instances within the cutoff time. More encouragingly, Table 5 shows that CScoreSAT solves the most k-SAT instances for each k, which illustrates its robustness. The superiority 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.

Effectiveness of Comprehensive Score
To demonstrate the effectiveness of the concept of comprehensive score, we test an alternative version of CScoreSAT that works without $cscore$ (i.e. with $cscore(x) = score(x)$) on random 5-SAT and 7-SAT instances. The alternative

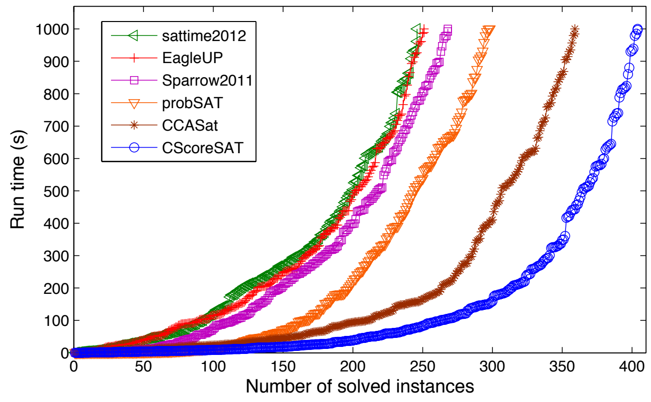

Figure 1: **Comparison of run time distributions on the SAT Challenge 2012 benchmark, with a cutoff time of 1000 seconds.**

algorithm fails to find a solution for any 5-SAT instance with $n > 2000$ or 7-SAT instance with $n > 200$. This indicates that the comprehensive score notion plays a key role in the CScoreSAT algorithm.

7 Summary and Future Work

This work took a significant step towards improving SLS algorithms for random k-SAT instances with long clauses. We combined a novel variable property called $subscore$ with the $score$ property, leading to a scoring function named comprehensive score. We also defined a new type of "decreasing" variables, namely comprehensively decreasing variables. These two significant notions lead to a new SLS algorithm called CScoreSAT, which is both efficient and robust. 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. Moreover, CScoreSAT significantly outperforms its competitors on random k-SAT instances with various clause-to-variable ratios for each $k = 4, 5, 6, 7$ from SAT Challenge 2012, indicating its robustness. Also, CScoreSAT is not sensitive to its parameters.

This work has opened up a new direction in improving SLS algorithms for SAT. A significant research issue is to improve SLS algorithms for structured instances by comprehensive scores. 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 973 Program 2010CB328103, ARC Grant FT0991785, National Natural Science Foundation of China (61073033, 61003056 and 60903054). We would like to thank the anonymous referees for their helpful comments.

References

[Achlioptas, 2009] Dimitris Achlioptas. Random satisfiability. In *Handbook of Satisfiability*, pages 245–270. 2009.

[Balint and Fröhlich, 2010] Adrian Balint and Andreas Fröhlich. Improving stochastic local search for SAT with a new probability distribution. In *Proc. of SAT-10*, pages 10–15, 2010.

[Balint and Schöning, 2012] Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In *Proc. of SAT-12*, pages 16–29, 2012.

[Balint *et al.*, 2010] Adrian Balint, Daniel Gall, Gregor Kapler, and Robert Retz. Experiment design and administration for computer clusters for SAT-solvers (EDACC). *JSAT*, 7(2-3):77–82, 2010.

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

[Cai and Su, 2012] Shaowei Cai and Kaile Su. Configuration checking with aspiration in local search for SAT. In *Proc. of AAAI-12*, pages 334–340, 2012.

[Cai and Su, 2013] Shaowei Cai and Kaile Su. Local search for boolean satisfiability with configuration checking and subscore. *submitted to Artif. Intell.*, 2013.

[Cai *et al.*, 2011] Shaowei Cai, Kaile Su, and Abdul Sattar. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artif. Intell.*, 175(9-10):1672–1696, 2011.

[Gableske and Heule, 2011] Oliver Gableske and Marijn Heule. EagleUP: Solving random 3-SAT using SLS with unit propagation. In *Proc. of SAT-11*, pages 367–368, 2011.

[Hoos and Stützle, 2004] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[Kautz *et al.*, 2009] Henry A. Kautz, Ashish Sabharwal, and Bart Selman. Incomplete algorithms. In *Handbook of Satisfiability*, pages 185–203. IOS Press, 2009.

[KhudaBukhsh *et al.*, 2009] Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Satenstein: Automatically building local search SAT solvers from components. In *Proc. of IJCAI-09*, pages 517–524, 2009.

[Kirkpatrick and Selman, 1994] Scott Kirkpatrick and Bart Selman. Critical behavior in the satisfiability of random boolean formulae. *Science*, 264:1297–1301, 1994.

[Kirkpatrick *et al.*, 1983] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Kroc *et al.*, 2010] Lukas Kroc, Ashish Sabharwal, and Bart Selman. An empirical study of optimal noise and runtime distributions in local search. In *Proc. of SAT-10*, pages 346–351, 2010.

[Li and Li, 2012] Chu Min Li and Yu Li. Satisfying versus falsifying in local search for satisfiability - (poster presentation). In *Proc. of SAT-12*, pages 477–478, 2012.

[Luo *et al.*, 2012] Chuan Luo, Kaile Su, and Shaowei Cai. Improving local search for random 3-sat using quantitative configuration checking. In *ECAI*, pages 570–575, 2012.

[Selman *et al.*, 1992] Bart Selman, Hector J. Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In *Proc. of AAAI-92*, pages 440–446, 1992.

[Selman *et al.*, 1994] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proc. of AAAI-94*, pages 337–343, 1994.

[Thornton *et al.*, 2004] John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for SAT. In *Proc. of AAAI-04*, pages 191–196, 2004.

[Tompkins and Hoos, 2010] Dave A. D. Tompkins and Holger H. Hoos. Dynamic scoring functions with variable expressions: New SLS methods for solving SAT. In *Proc. of SAT-10*, pages 278–292, 2010.

[Tompkins *et al.*, 2011] Dave A. D. Tompkins, Adrian Balint, and Holger H. Hoos. Captain jack: New variable selection heuristics in local search for SAT. In *Proc. of SAT-11*, pages 302–316, 2011.