Artificial Intelligence 204 (2013) 75–98

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Local search for Boolean Satisfiability with configuration checking and subscore

Shaowei Cai [a,b,*], Kaile Su [b,c]

[a] Queensland Research Lab, NICTA, Brisbane, Australia
[b] Key Laboratory of High Confidence Software Technologies, Peking University, Beijing, China
[c] IIIS, Griffith University, Brisbane, Australia

ARTICLE INFO

Article history:
Received 10 December 2012
Received in revised form 1 September 2013
Accepted 1 September 2013
Available online 9 September 2013

Keywords:
SAT
Local search
Configuration checking
Subscore

ABSTRACT

This paper presents and analyzes two new efficient local search strategies for the Boolean Satisfiability (SAT) problem. We start by proposing a local search strategy called configuration checking (CC) for SAT. The CC strategy results in a simple local search algorithm for SAT called Swcc, which shows promising experimental results on random 3-SAT instances, and outperforms TNM, the winner of SAT Competition 2009.

However, the CC strategy for SAT is still in a nascent stage, and Swcc cannot yet compete with Sparrow2011, which won SAT Competition 2011 just after Swcc had been designed. The CC strategy seems too strict in that it forbids flipping those variables even with great scores, if they do not satisfy the CC criterion. We improve the CC strategy by adopting an aspiration mechanism, and get a new variable selection heuristic called configuration checking with aspiration (CCA). The CCA heuristic leads to an improved algorithm called Swcca, which exhibits state-of-the-art performance on random 3-SAT instances and crafted ones.

The third contribution concerns improving local search algorithms for random k-SAT instances with $k > 3$. Although the SAT community has made great achievements in solving random 3-SAT instances, the progress lags far behind on random k-SAT instances with $k > 3$. This work proposes a new variable property called subscore, which is utilized to break ties in the CCA heuristic when candidate variables for flipping have the same score. The resulting algorithm CCAsubscore is very efficient for solving random k-SAT instances with $k > 3$, and significantly outperforms other state-of-the-art ones. Combining Swcca and CCAsubscore, we obtain a local search SAT solver called CCASat, which was ranked first in the random track of SAT Challenge 2012.

Additionally, we perform theoretical analyses on the CC strategy and the subscore property, and show interesting results on these two heuristics. Particularly, our analysis indicates that the CC strategy is more effective for k-SAT with smaller k, while the subscore notion is not suitable for solving random 3-SAT.

1. Introduction

The Boolean Satisfiability problem (SAT) is a prototypical NP-complete problem. It is central to many domains of computer science and artificial intelligence, and has been widely studied due to its significant importance in both theory and applications [23]. Stochastic local search (SLS) algorithms are among the best known methods currently available for solving

* Corresponding author.
E-mail addresses: shaoweicai.cs@gmail.com (S. Cai), k.su@griffith.edu.au (K. Su).

0004-3702/$ – see front matter
http://dx.doi.org/10.1016/j.artint.2013.09.001

certain types of SAT instances. SLS algorithms are typically incomplete, i.e., they cannot determine with certainty that a given Boolean formula is unsatisfiable. However, they often find models of satisfiable formulas surprisingly effectively [21].

SLS is well known as the most effective approach for solving random satisfiable instances, and SLS algorithms are often evaluated on 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 (either good or bad) on random k-SAT instances. 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.

The basic schema for an SLS algorithm for SAT is as follows. Beginning with a random complete assignment of truth values to variables, in each subsequent search step the algorithm chooses a variable and flips it. We use *pickVar* to denote the function for selecting the variable to be flipped.

As stated in [47], SLS algorithms for SAT usually work in two different modes, i.e., the greedy (intensification) mode and the diversification mode. In the greedy mode, the *pickVar* functions prefer variables whose flips can decrease the number of falsified clauses (or the total weight of falsified clauses in clause weighting SLS algorithms); 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 such as *age* and *flip count* to pick a variable for this aim.

An important issue for local search is the cycling problem, i.e., revisiting a candidate solution that has been visited recently [36]. More generally, the cycling problem refers to the phenomenon that a local search algorithm spends too much time in searching a small part of search space. A strategy called configuration checking (CC) was recently proposed to address this issue, and was used to improve a state-of-the-art Minimum Vertex Cover (MVC) local search algorithm called EWLS [10], leading to the much more efficient local search algorithm EWCC [12]. A natural question is whether the CC idea also works for SAT.

As the first contribution, this work proposes a CC strategy for SAT, which forbids a variable x to be flipped if none of its neighboring variables has been flipped since the last time x was flipped. We note that the CC strategy is novel in local search algorithms for SAT. Although the CC strategies for MVC [12] and SAT share the same intuition of reducing local structure cycles, they differ in technical aspects, such as the definition of configuration and the checking mechanism. The CC strategy for SAT is used in developing a local search algorithm for SAT called Swcc (Smoothed Weighting with Configuration Checking).

Swcc shows promising performance on random 3-SAT instances and outperforms TNM [28], the winner of the random satisfiable category of SAT Competition 2009. However, the CC strategy for SAT is still in a nascent stage, and Swcc cannot yet compete with Sparrow2011 [3], which won the random satisfiable category of SAT Competition 2011 just after Swcc had been designed. In our opinion, the CC strategy for SAT is too strict for the local search to quickly recover from a bad flip with a big loss, which may occur in the diversification mode. This is because the CC strategy forbids flipping any variable whose configuration (i.e., the truth values of all its neighboring variables) has not been changed since its last flip, regardless of the benefit its flip can bring.

The second contribution of this work is to improve the CC strategy for SAT, by remedying its shortcoming above. To do so, we combine an aspiration mechanism with the CC strategy to flex it, leading to a new *pickVar* heuristic called Configuration Checking with Aspiration (CCA). Note that the aspiration idea in CCA is inspired by the aspiration mechanisms in tabu search [19,41]. According to CCA, there are two levels with different priorities in the greedy mode. Those variables whose flips can bring a big benefit have a chance to be selected on the second level, even if they do not satisfy the CC criterion. The CCA heuristic is used to improve Swcc, resulting in a new local search algorithm called Swcca (Smoothed Weighting and Configuration Checking with Aspiration).

Significantly improving Swcc, Swcca achieves state-of-the-art performance on random 3-SAT instances as well as crafted ones. Our experiments show that Swcca is 2–3 times faster than Sparrow2011 on large random 3-SAT instances. Also, the experiments on crafted instances demonstrate that Swcca is competitive with sattime [27], which is the best local search solver for the crafted satisfiable category of SAT Competition 2011.

The third contribution concerns improving local search algorithms for random k-SAT instances with $k > 3$. Although the SAT community has made great achievements in solving random 3-SAT instances in the past two decades, the progress lags far behind on random k-SAT instances with $k > 3$. In this work, we propose a new variable property called *subscore*, which is related to but different from the commonly used variable property *score*. While the *score* property considers the transformations between satisfied and falsified clauses by flipping a variable, the *subscore* property takes into account the transformations between *critical clauses* (i.e., those clauses having exactly one true literal) and the clauses with two true literals. In some sense, the subscore property can be seen as an extension of the score property. Using subscore to break ties among candidate variables with the equally greatest score in the CCA heuristic, we design a new local search algorithm for SAT called CCAsubscore.

CCAsubscore exhibits outstanding performance for random k-SAT with $k > 3$, significantly better than previous SLS algorithms, including the three winners in SAT Competition 2011, namely Sparrow2011, sattime2011 [27] and EagleUP [15]. Combining the Swcca and CCAsubscore algorithms, we obtain a local search SAT solver called CCASat, which was ranked

first in the random track of SAT Challenge 2012.[1] The codes of all the solvers developed in this paper, including Swcc, Swcca and CCASat, are publicly available online.[2]

Finally, we perform some theoretical analyses on the configuration checking strategy for SAT and the subscore property. We analyze the relationships of the CC strategy with the widely used promising decreasing variable exploiting strategy [26] and the tabu method in terms of forbidding strength. Moreover, as the CC strategy is not always effective, we predict when it is effective through both theoretical and experimental analysis. Generally, the effectiveness of the CC strategy decreases as k increases. Specifically, our analysis shows that the CC strategy is effective for random k-SAT instances with $k < 6$ and ineffective for those with $k \geqslant 6$. We also show that the subscore based tie-breaking mechanism is ineffective for random 3-SAT instances, although it is very effective for random k-SAT instances with $k > 3$.

We note that the first contribution (Section 3) and the second contribution (Section 4) have been published in [8] and [9] respectively, while the third contribution (Section 5), as well as further experiments and analyses (Sections 6 and 7) are new in this paper.

The remainder of this paper is organized as follows. Some definitions and notations are given in the next section. Section 3 proposes the CC strategy for SAT and the Swcc algorithm, along with the experiments on Swcc. Section 4 presents the CCA *pickVar* heuristic and the Swcca algorithm, as well as the experiments on Swcca. Section 5 introduces the notion of subscore and presents the CCAsubscore algorithm, followed by experiments on CCAsubscore. In Section 6, we demonstrate the performance of CCASat by summarizing and analyzing the experimental results of the random track of SAT Challenge 2012. Further analyses on the CC strategy and the subscore property are carried out in Section 7. Finally, we give some concluding remarks and future directions.

2. Preliminaries

Given a set of n Boolean *variables* $\{x_1, x_2, \ldots, x_n\}$, 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 $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is a conjunction of clauses. The Boolean Satisfiability problem (SAT) consists in testing whether all clauses in a given CNF formula F can be satisfied by some consistent assignment of truth values to variables.

A well-known generation model for SAT is the random k-SAT model [1]. A random k-SAT formula with n variables and m clauses, denoted by $F_k(n, m)$, is a CNF formula where the clauses are chosen uniformly, independently and without replacement among all $2^k\binom{n}{k}$ non-trivial clauses of length k, i.e., clauses with k distinct, non-complementary literals.

For a formula F, we use $V(F)$ to denote the set of all variables that appear in F, and $r = m/n$ to denote its (clause-to-variable) *ratio*. We say a variable x occurs in a clause, if the clause contains either x or $\neg x$. Two variables are neighbors if and only if they occur simultaneously in at least one clause. The neighborhood of a variable x is $N(x) = \{y | y$ occurs in at least one clause with $x\}$, which is the set of all *neighboring variables* of variable x.

A mapping $\alpha : V(F) \to \{0, 1\}$ is called an *assignment*. If α maps all variables to a Boolean value, it is *complete*. For local search algorithms for SAT, a candidate solution is a complete assignment. Usually, a local search algorithm for SAT starts from a complete assignment, and flips a variable iteratively to search for a satisfying assignment. In each step, the variable to be flipped is selected from a certain set of variables, and each variable in that set is called a *candidate variable*.

If a literal evaluates to true under the given assignment, we say it is a *true literal*. Otherwise, we say it is a *false literal*. The set of states of a clause is {*satisfied*, *falsified*}. A clause is *satisfied* if it has at least one true literal under the given assignment, and *falsified* if it has no true literal under the given assignment.

Generally, for a formula F, we use $cost(F, \alpha)$ to denote the number of falsified clauses under an assignment α. In dynamic local search algorithms which use clause weighting techniques, however, $cost(F, \alpha)$ denotes the total weight of all falsified clauses under an assignment α. The variable property *score* is defined as $score(x) = cost(F, \alpha) - cost(F, \alpha')$, where α' is obtained from α by flipping x. A variable x is *decreasing* if and only if $score(x) > 0$, as its flip will decrease the *cost*. 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, by flipping x. In dynamic local search algorithms, *make* and *break* measures the total weight of clauses that would become satisfied and falsified, respectively, by flipping x. Note that the two definitions of *score* are equivalent. The *age* of a variable is defined as the number of search steps that have occurred since the variable was last flipped.

We would like to note that all SLS algorithms presented in this paper, including Swcc, Swcca, CCAsubscore and CCASat, utilize clause weighting techniques, and thus use the weighted version of *score*.

3. Configuration checking and the Swcc algorithm

In this section, we propose the configuration checking (CC) strategy for SAT and use it to develop an SLS algorithm called Swcc (Smoothed Weighting with Configuration Checking). Then we carry out experiments to evaluate the performance of Swcc on random 3-SAT instances from SAT Competition 2009, which was the latest SAT competition when Swcc was developed.

[1] http://baldur.iti.kit.edu/SAT-Challenge-2012/results.html.
[2] http://www.shaoweicai.net/SAT.html.

3.1. Configuration checking

Originally introduced in [12], configuration checking (CC) is a diversification strategy aiming to reduce the cycling problem in local search. The intuition behind this idea is that by reducing cycles on local structures of the candidate solution, we may reduce cycles on the whole candidate solution.

3.1.1. A configuration checking strategy for SAT and CCD variables

The CC strategy is based on the concept of *configuration*. In this work, the configuration of a variable refers to truth values of all its neighboring variables. The formal definition is given as follows:

Definition 1. Given a CNF formula F and an assignment α to $V(F)$, the *configuration* of a variable $x \in V(F)$ under α is a vector $Conf_\alpha(x)$ consisting of truth values of all variables in $N(x)$ under α, i.e., $Conf_\alpha(x) = \alpha|_{N(x)}$, which is the assignment restricted to $N(x)$.

When we are talking about the configuration of variables, the assignment α is usually explicit from the context, and thus omitted.

Given a SAT local search algorithm solving a CNF formula F, the configuration checking strategy can be described as follows. For a variable $x \in V(F)$, if the configuration of x has not been changed (in this work, this means none of its neighboring variables has been flipped) since its last flip, then it is forbidden to be flipped. Typically, the CC strategy is used in the greedy mode, to decrease blind unreasonable greedy search.

Based on the CC strategy, it is natural to define the concept of configuration changed decreasing variables, which are considered to be good candidate variables for flipping and should be preferable than other variables.

Definition 2. A *configuration changed decreasing* (CCD) variable is a decreasing variable whose configuration has been changed since its last flip.

The concept of CCD variables is of significant importance to the algorithms proposed in this work, as we will present.

3.1.2. An implementation of the CC strategy

To implement the CC strategy, we employ an array *confChange*, whose element is an indicator for a variable — $confChange(x) = 1$ means the configuration of variable x has been changed since the last time x was flipped; and $confChange(x) = 0$ on the contrary. During the search procedure, the variables with a *confChange* value of 0 are forbidden to be flipped. The *confChange* array is initialized by setting $confChange(x)$ to 1 for each variable x. After that, when flipping a variable x, $confChange(x)$ is reset to 0, and for each variable $y \in N(x)$, $confChange(y)$ is set to 1.

With the above implementation of the CC strategy, a variable x is a CCD variable if and only if $score(x) > 0$ and $confChange(x) = 1$ hold at the same time.

3.1.3. Remarks on the CC strategy

Previous local search algorithms for SAT usually select a variable to flip based on variable properties such as *score* [20], *break* [43] and *age* [17]. Some of them also utilize dynamic score [48] and scoring functions which combine different variable properties [3,47]. The CC strategy takes into account the variables' circumstance information (which refers to the truth value of neighboring variables in this work) when selecting a variable to flip. It appears reasonable and helpful to incorporate such a circumstance-concerning strategy to the traditional variable-based heuristics, as the best decision on a variable should come from not only its own information, but also its circumstance.

We would like to stress that configuration checking is a general algorithmic idea for local search, more than a specific strategy. By defining different forms of configuration and checking mechanism, one can obtain various CC heuristics. For instance, in a variant of the CC strategy named Quantitative Configuration Checking [31], the configuration of a variable x is defined as the state of the clauses in which x appears, and the checking mechanism is performed in a quantitative way.

3.2. The Swcc algorithm

In this subsection, we utilize the CC strategy to develop a new SLS algorithm for SAT called Swcc.

The SWT scheme: Swcc employs a clause weighting scheme which resembles in some respect the SAPS scheme [22], and is referred to as SWT, short for Smooth Weighting based on Threshold. Whenever SWT is called, it works as follows. It increases clause weights of all falsified clauses by one; further, if the averaged weight $\overline{w}$ exceeds a threshold γ, it smooths all clause weights as $w(c_i) := \lfloor \rho \cdot w(c_i) \rfloor + \lfloor (1-\rho)\overline{w} \rfloor$, where $0 < \rho < 1$.

The Swcc algorithm is outlined in Algorithm 1, as described below. In the beginning, the algorithm generates a random complete assignment α, initializes all clause weights as 1 and computes *scores* of variables accordingly, and initializes $confChange(x)$ as 1 for each variable x.

After initialization, Swcc executes a loop until it finds a satisfying assignment or reaches the time limit. In each step, Swcc works in either the greedy mode or the diversification mode, depending on the existence of CCD variables.

Algorithm 1: Swcc

```
Input: CNF-formula F, maxSteps
Output: A satisfying assignment α of F, or "no solution found"
1  begin
2      α := randomly generated truth assignment;
3      for step := 1 to maxSteps do
4          if α satisfies F then return α;
5          if there exist decreasing variables whose configuration has been changed after its last flip then
6              v := such a variable with the greatest score, breaking ties in favor of the oldest one;
7          else
8              update clause weights according to the SWT scheme;
9              v := the oldest variable in a random falsified clause;
10         α := α with v flipped;
11     return "no solution found";
12 end
```

The greedy mode: If there exist CCD variables, Swcc selects a CCD variable with the greatest *score* to flip, breaking ties in favor of the oldest variable.

The diversification mode: If no CCD variable is present, then Swcc switches to the diversification mode. Specifically, it first updates clause weights according to the SWT scheme. Then, it chooses a uniform random falsified clause, and picks the oldest variable in that clause to flip.

3.3. Data structures for CCD variables

The complexity of the CC strategy mainly depends on the implementation of the data structures for CCD variables. We employ a stack named CCDVar to store all CCD variables. Moreover, we utilize an auxiliary array *recorded* for avoiding duplication in the CCDVar stack. Each element of the array "recorded" is an indicator for a variable — $recorded(x) = 1$ means x is already stored in CCDVar; and $recorded(x) = 0$ on the contrary.

In the beginning, all variables with $score(x) > 0$ and $confChange(x) = 1$ are pushed into the CCDVar stack, and $recorded(x)$ is set to 1 for these variables and 0 for the others. During the search procedure, we update the CCDVar stack, and along with updating CCDVar, the array *recorded* is updated accordingly: when pushing a variable x into CCDVar, $recorded(x)$ is set to 1; when removing a variable x out of CCDVar, $recorded(x)$ is set to 0. In dynamic local search algorithms, such as the algorithms in this work, the CCDVar stack is updated in two cases: (a) flipping a variable and (b) updating clause weights.

(a) Updating the CCDVar stack when flipping a variable: When flipping a variable, the CCDVar stack is updated in a two-phase process. First, those variables that no longer satisfy the conditions ($score(x) > 0$ and $confChange(x) = 1$) are removed from CCDVar; secondly, those variables currently not in CCDVar but will satisfy the conditions after the flip are pushed into CCDVar. In the "removing" stage, we scan the CCDVar stack and remove those variables whose scores are no longer positive from it. We argue that by doing so we remove all the variables that no longer satisfy the conditions from CCDVar. This is because in each step the only variable whose confChange value changes from 1 to 0 is the flipped variable, and if the flipped variable x was in CCDVar before the flip, then $score(x) < 0$ holds after variable x being flipped (flipping x would make $score(x)$ be its opposite number); in this case, the flipped variable x will be removed since its score is negative. In the "adding" stage, we scan the neighboring variables of the flipped variable, and push those with $score(y) > 0$ and $recorded(y) = 0$ into the CCDVar stack. Here we do not check the condition $confChange(y) = 1$ because for each neighboring variable y of the flipped variable, $confChange(y)$ is set to 1 according to the updating rule of the confChange array.

(b) Updating the CCDVar stack when updating clause weights: When increasing the clause weight of a falsified clause by one, the scores of all variables in the clause are also increased by 1. If this updating makes the score of a variable x become positive from non-positive, and at that moment $confChange(x) = 1$ holds, then we push x into the CCDVar stack.

3.4. Evaluations of Swcc

In this subsection, we carry out experimental studies to evaluate the performance of Swcc on random 3-SAT instances. Also, in order to demonstrate the effectiveness of the CC strategy, we compare Swcc with its three alternatives, two of which replace the CC strategy with the tabu method and the promising decreasing strategy respectively, and the third one just removes the CC strategy.

3.4.1. Benchmarks and experiment preliminaries

We evaluate Swcc on the **3-SAT Comp09** benchmark, which comprises all large random 3-SAT instances from SAT Competition 2009 ($r = 4.2$, $2000 \leqslant n \leqslant 26000$, 10 instances each size).

Swcc is implemented in C++ and compiled by g++ with the '-O2' option. For the two parameters in Swcc, after trying all combinations of $\gamma = 100, 200, \ldots, 1000$ (the performance of Swcca degrades significantly when γ exceeds 1000) and $\rho = 0.1, 0.2, \ldots, 0.9$, we set $\gamma = 300$ and $\rho = 0.3$, as this setting yields the best performance among all the settings. We

Table 1
Comparative performance results of Swcc, TNM and Sparrow2011 on the 3-SAT Comp09 benchmark. Each solver is run 100 times on each instance with a cutoff time of 1000 seconds.

Instance class	TNM		Sparrow2011		Swcc	
	suc rate	avg time	suc rate	avg time	suc rate	avg time
3SAT-v2000	100%	1.6	100%	1.4	100%	1.4
3SAT-v4000	94.9%	74	98.5%	48	100%	21.6
3SAT-v6000	99.5%	42	100%	17	100%	43
3SAT-v8000	98.5%	63	99.6%	34	99.1%	56
3SAT-v10000	99.8%	64	100%	14	100%	29
3SAT-v12000	97.1%	194	100%	46	100%	93
3SAT-v14000	94.2%	291	100%	49	100%	109
3SAT-v16000	96.5%	232	100%	29	100%	62
3SAT-v18000	92.3%	351	100%	37	100%	81
3SAT-v20000	45.6%	827	94.1%	236	92%	326
3SAT-v22000	59.8%	735	99.8%	108	99.3%	207
3SAT-v24000	59.0%	707	96.3%	158	93.5%	215
3SAT-v26000	55.8%	686	99.6%	106	97.5%	195

also try the parameters near $\gamma = 300$ and $\rho = 0.3$ in a higher degree of accuracy, trying to find a more optimal setting for Swcc, but do not observe any noticeable improvement. This also indicates Swcc is not so sensitive to its parameters.

We compare Swcc with two SLS solvers TNM [28] and Sparrow2011 [3], which won the gold medal of the random satisfiable category of SAT Competition 2009 and 2011 respectively. Particularly, Sparrow2011 significantly outperformed other competitors on random 3-SAT instances. The solvers TNM and Sparrow2011 in our experiments are the ones submitted to SAT Competition 2009 and 2011 respectively.

Experiments in this section are run on a machine with a 3 GHz Intel Core E8400 CPU and 4 GB RAM under Linux. We run each solver 100 times for each instance and thus 1000 times for each instance class, with a cutoff time of 1000 seconds. We report the success rate ("suc rate"), i.e., the number of successful runs divided by the number of total runs, as well as the averaged run time in CPU seconds ("avg time") for each solver on each instance class.

3.4.2. Comparing Swcc with TNM and Sparrow2011 on random 3-SAT

Table 1 summarizes the performance of Swcc on the 3-SAT Comp09 benchmark, compared with TNM and Sparrow2011. The results show that Swcc outperforms TNM significantly on these instances. Specially, for the instances with at least 20 000 variables, Swcc finds a satisfying solution in nearly every run, while TNM only succeeds in half of the runs, which indicates a significant performance gap between TNM and Swcc. On the other hand, Swcc cannot yet rival Sparrow2011 on these random 3-SAT instances, and the averaged run time of Swcc is about 1.5 to 2 times that of Sparrow2011.

3.4.3. Effectiveness of the CC strategy on random 3-SAT

To demonstrate the effectiveness of the CC strategy in Swcc, we first compare Swcc with its two alternative algorithms, which replace the CC strategy with the tabu mechanism and the promising decreasing strategy, respectively. We also evaluate Swcc without the CC strategy to study how much the CC strategy contributes to Swcc.

The tabu mechanism [18,19] is a previous significant method for handling the cycling problem in local search, and has been widely used in local search algorithms [32,6,44,16]. To prevent the local search from immediately returning to a previously visited candidate solution, the tabu mechanism forbids reversing the recent changes, where the forbidding strength is controlled by a parameter called *tabu tenure* (tt).

The promising decreasing strategy [26] is a previous significant strategy for improving the greedy mode of SLS algorithms for SAT. Originally proposed in the G^{2}WSAT algorithm [26], it has been widely applied in local search SAT solvers. This strategy prefers to flip the best (with respect to score) promising decreasing variable in the greedy mode if such variables exist. Roughly speaking, a decreasing variable is promising decreasing if and only if it becomes decreasing (directly) because of another variable's flip.

We modify Swcc to obtain two alternative algorithms called Swtabu and Swprm by replacing the CC strategy with the tabu mechanism and the promising decreasing strategy respectively. Both Swtabu and Swprm follow the same procedure as Swcc, except for one modification: In the greedy mode, Swtabu selects the best variable (with respect to score) from the decreasing variables whose *ages* are more than tt, and Swprm selects the best promising decreasing variable (with respect to score).

Table 2 summarizes the comparative performance results of Swcc, Swtabu and Swprm on the 3-SAT Comp09 benchmark. Swtabu is performed with the optimal value of the tt parameter ("opt tt" in Table 2) for each instance class. The optimal value of tt is obtained manually by trying tt from 1 to 100.

As can be seen from Table 2, Swprm performs essentially worse than the other two solvers on these random 3-SAT instances. This is because there are too few promising decreasing variables during the search, and when there are not such variables, the algorithm works in the diversification mode, which is purely diversified. This makes the search bias too much towards diversification. Note that G^{2}WSAT and other SLS solvers utilizing the promising decreasing strategy employ

Table 2
Comparative performance results of Swcc and its two alternatives, namely Swtabu (with optimal tabu tenure) and Swprm, on the 3-SAT Comp09 benchmark. Each solver is performed 100 times on each instance with a cutoff time of 1000 seconds.

Instance class	Swtabu			Swprm		Swcc	
	opt tt	suc rate	avg time	suc rate	avg time	suc rate	avg time
3SAT-v2000	6	100%	1.3	100%	5.3	100%	1.3
3SAT-v4000	12	100%	20	87.8%	198.7	100%	21.6
3SAT-v6000	13	100%	45	56.9%	558	100%	43
3SAT-v8000	15	93.3%	85	35.0%	783	**99.1%**	**56**
3SAT-v10000	18	100%	106	13.2%	930	**100%**	**29**
3SAT-v12000	20	90.7%	339	0.4%	998	**100%**	**93**
3SAT-v14000	25	85%	522	1.4%	993	**100%**	**109**
3SAT-v16000	24	84.9%	462	0%	1000	**100%**	**62**
3SAT-v18000	28	72.3%	626	0%	1000	**100%**	**81**
3SAT-v20000	30	4.3%	962	0%	1000	**92%**	**326**
3SAT-v22000	38	5.9%	973	0%	1000	**99.3%**	**207**
3SAT-v24000	30	4.1%	989	0%	1000	**93.5%**	**215**
3SAT-v26000	36	5.2%	973	0%	1000	**97.5%**	**195**

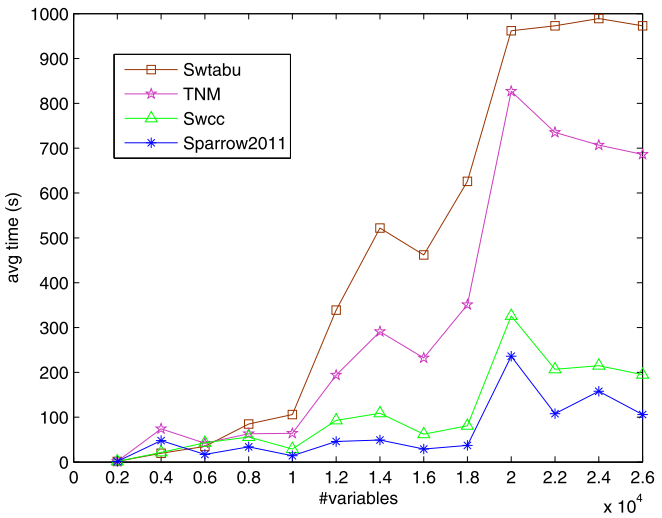

Fig. 1. Comparing averaged run time of Swcc with Swtabu, TNM and Sparrow2011 on the 3-SAT Comp09 benchmark, with a cutoff time of 1000 seconds.

Novelty-like heuristics [34] when there are not promising decreasing variables, which still prefer to pick the variable with greater scores, and thus can make a good balance between intensification and diversification.

Now we focus on the comparison between Swcc and Swtabu. The two algorithms perform similarly on small instances with $n \leqslant 6000$. However, for the larger instances, the performance of Swcc is much better than that of Swtabu. Specially, for the instances with $n \geqslant 20000$, the success rates of Swcc are very close to 100%, while those of Swtabu are always less than 10%. These experimental results suggest that the CC strategy is more promising than the tabu mechanism in solving random 3-SAT instances. Fig. 1 demonstrates how the averaged run time of Swcc and Swtabu vary with increasing problem size, along with that of TNM and Sparrow2011.

In order to study how much the CC strategy contributes to Swcc, we also run another alternative of Swcc which works without the CC strategy on the 3-SAT Comp09 benchmark. This alternative algorithm works the same as Swcc, except for that in the greedy mode, it chooses the best decreasing variable (with respect to score), rather than the best CCD variable. Surprisingly, our experiments show that this alternative solver fails to find a solution for any instance with $n > 4000$. This indicates that the CC strategy plays a key role in the Swcc algorithm.

4. Configuration checking with aspiration and the Swcca algorithm

In this section, we improve the CC strategy by combining it with an aspiration mechanism, which results in a new *pickVar* heuristic called Configuration Checking with Aspiration (CCA). Then we utilize the CCA heuristic to enhance Swcc, leading to a new SLS algorithm called Swcca. Our experiments show that Swcca exhibits state-of-the-art performance on random 3-SAT instances, and also has promising performance on crafted ones.

4.1. Configuration checking with aspiration

Although the CC strategy shows its effectiveness in local search algorithms for SAT, it is still in its infancy. We consider the CC strategy for SAT is too strict for the local search to quickly recover from a bad flip with a big loss, which may occur in the diversification mode. This is because the CC strategy forbids flipping any variable whose configuration has not been changed since its last flip, regardless of the benefit its flip can bring. This lack of differentiation is a serious disadvantage in our opinion.

To overcome this drawback, we combine an aspiration mechanism with the CC strategy to flex it. Also, to get a complete *pickVar* heuristic (recalling that CC only works in the greedy mode), we specify the diversification mode, which is so simple that it always picks the oldest variable from a random falsified clause, as Swcc does. The resulting *pickVar* heuristic is called Configuration Checking with Aspiration (CCA).

Before getting into the details of the CCA heuristic, we first give a definition. A variable x is a *significant decreasing* (SD) variable if $score(x) > g$, where g is a positive integer large enough. In this work, g is always set to the averaged clause weight (over all clauses) $\overline{w}$.

The CCA heuristic (Algorithm 3) switches between the greedy mode and the diversification mode, as described below. In the greedy mode, there are two levels with descending priorities. On the first level it does a gradient walk, i.e., picking a CCD variable with the greatest score to flip. If no CCD variable is present, CCA activates the "aspiration criterion" to go to the second level, where it selects an SD variable with the greatest score to flip if any. If there exists neither CCD nor SD variable, the CCA heuristic switches to the diversification mode, where the oldest variable in a uniformly random falsified clause is picked to flip.

Algorithm 2: *pickVar*-heuristic CCA

```
1 if ∃ CCD variables then return a CCD variable with the greatest score;
2 if ∃ SD variables then return an SD variable with the greatest score;
3 return the oldest variable in a random falsified clause;
```

The aspiration mechanism renders the CC strategy more flexible by selecting the variables with great scores to flip, as explained below. In the diversification mode, SLS algorithms may flip a variable whose score is a negative integer with a large absolute value. These variables are forbidden to be flipped by the CC strategy until one of their neighboring variables is flipped. Without the aspiration mechanism, such variables which are "mistakenly" flipped in the diversification mode will accumulate. This would delay the local search transferring to promising regions of search space. The aspiration mechanism corrects such mistakes and thus helps the local search transfer to promising regions in time.

Exactly speaking, the CCA heuristic is a local search framework, rather than a specific heuristic. To make it a specific heuristic, one needs to specify the tie-breaking mechanisms in the greedy mode. Also, one can modify the variable selection heuristic in the random mode, and various techniques such as clause weighting schemes can be used in this framework.

4.2. The Swcca algorithm

We employ the CCA heuristic to improve the Swcc algorithm, resulting in a new SLS algorithm called Swcca. For the tie-breaking mechanism in the CCA heuristic, Swcca breaks ties by preferring the oldest variable, as Swcc does.

Algorithm 3: Swcca

```
   Input: CNF-formula F, maxSteps
   Output: A satisfying assignment α of F, or "no solution found"
1  begin
2      α := randomly generated truth assignment;
3      for step := 1 to maxSteps do
4          if α satisfies F then return α;
5          if there exist decreasing variables whose configuration has been changed after its last flip then
6              v := such a variable with the greatest score, breaking ties in favor of the oldest one;
7          else if there exist variables with a score higher than w̄ then
8              v := such a variable with the greatest score, breaking ties in favor of the oldest one;
9          else
10             update clause weights according to the SWT scheme;
11             v := the oldest variable in a random falsified clause;
12         α := α with v flipped;
13     return "no solution found";
14 end
```

Table 3
Comparative performance results of Swcc, Swcca and Sparrow2011 on the 3-SAT Comp11 benchmark. Each solver is performed 100 times on each instance with a cutoff time of 1000 seconds.

Instance class	Swcc		Sparrow2011		Swcca	
	suc rate	avg time	suc rate	avg time	suc rate	avg time
3SAT-v2500	100%	29	99.5%	30	100%	13
3SAT-v5000	100%	55	100%	19	100%	14
3SAT-v10000	99.4%	75	99.9%	40	100%	26
3SAT-v15000	99.1%	125	100%	57	100%	41
3SAT-v20000	98.5%	198	99.9%	98	100%	62
3SAT-v25000	96.9%	283	98.9%	169	100%	89
3SAT-v30000	93.5%	358	97.9%	208	100%	106
3SAT-v35000	80.3%	545	92.4%	360	100%	186
3SAT-v40000	81.8%	496	90.7%	321	99.9%	162
3SAT-v50000	51.2%	778	67.8%	584	100%	262

The Swcca algorithm is outlined in Algorithm 3. Swcca switches between the greedy mode and the diversification mode (as with Swcc), and differs from Swcc only in the greedy mode. Specifically, Swcca works as follows in the greedy mode. If there exist CCD variables, Swcca selects the CCD variable with the greatest *score* to flip, breaking ties in favor of the oldest one. If no CCD variable exists, Swcca picks the SD variable with the greatest *score* to flip if any, breaking ties in favor of the oldest one.

The SD variables are identified by checking all variables in falsified clauses, as any variable absent in falsified clauses is impossible to be decreasing. We do not use more "clever" implementations for maintaining the set of SD variables because it is not used that frequently compared to the set of CCD variables, according to our experiments. For other data structures in Swcca, we adopt the same implementations as in Swcc.

4.3. Evaluations of Swcca

In this subsection, we carry out experiments to evaluate the performance of Swcca on random 3-SAT instances and structured ones.

4.3.1. Benchmarks and experiment preliminaries

To evaluate Swcca, we set up four benchmarks.

1. **3-SAT Comp11**: all large random 3-SAT instances from SAT Competition 2011 ($r = 4.2$, $2500 \leqslant n \leqslant 50\,000$, 10 instances for each size).
2. **3-SAT Huge**: 600 satisfiable 3-SAT instances generated randomly ($r = 4.2$, $55\,000 \leqslant n \leqslant 80\,000$, 100 instances for each size). These huge instances are for measuring the superiority of Swcca on random 3-SAT instances more accurately.
3. **Crafted Benchmark**: the selected benchmark of crafted instances in SAT Competition 2011.[3] All unsatisfiable instances are removed, resulting in 84 instances in this benchmark.
4. **Application Benchmark**: the selected benchmark of application instances in SAT Competition 2011.[3] All unsatisfiable instances are removed, resulting in 78 instances in this benchmark.

Swcca is implemented in C++ and compiled by g++ with the '-O2' option. We set $\gamma = 300$ and $\rho = 0.3$ for the SWT scheme, as with Swcc. For random 3-SAT instances, we compare Swcca with Swcc and Sprrow2011; for crafted and application instances, we compare Swcca with sattime [27] and glucose [2]. Note that sattime is the best SLS solver for the crafted satisfiable category of SAT Competition 2011,[4] and beats all complete (CDCL-based) algorithms, which have been considered more effective than SLS algorithms for structured SAT problems for a long time. The complete solver glucose is developed based on MiniSAT [13], and is the best single-engine solver in the Application track of SAT Challenge 2012, and also the winner of SAT + UNSAT Application category in SAT Competition 2011. The solvers Sparrow2011, sattime and glucose (version 2) in our experiments are the ones submitted to SAT Competition 2011.

Experiments in this section are run on a machine with a 3 GHz Intel Core E8400 CPU and 4 GB RAM under Linux (the same as in Section 3). The cutoff time is set to 1000 seconds for the 3-SAT Comp11 benchmark, and 1800 seconds (half an

[3] http://www.cril.univ-artois.fr/SAT11/bench/SAT11-Competition-SelectedBenchmarks.tar.
[4] http://www.cril.univ-artois.fr/SAT11/results/ranking.php?idev=46&cpuLimit=5000&wcLimit=5000.

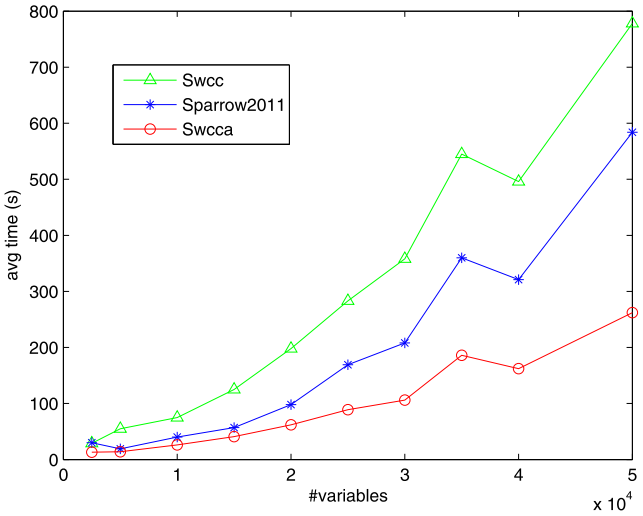

Fig. 2. Comparing averaged run time of Swcca with Swcc and Sparrow2011 on the 3-SAT Comp11 benchmark, with a cutoff time of 1000 seconds.

Table 4
Comparative performance results of Swcca and Sparrow2011 on the 3-SAT Huge benchmark. Each solver is run 100 times on each instance with a cutoff time of 1800 seconds.

Instance class	Sparrow2011			Swcca		
	suc rate	avg time	#flips(10^6)	suc rate	avg time	#flips(10^6)
3SAT-v55000	94.8%	591	380.8	100%	207	98.6
3SAT-v60000	88%	730	456.6	100%	248	112.6
3SAT-v65000	87.8%	843	511.6	100%	294	128.7
3SAT-v70000	85.6%	880	516.3	99%	401	157.4
3SAT-v75000	74.6%	1078	620.1	99.4%	477	189.1
3SAT-v80000	68%	1187	658.1	97.2%	631	242.0

hour) for the other three benchmarks. For each instance, each SLS solver is performed 100 times (except for the application instances where each SLS solver is performed only one run), while the complete solver glucose is performed one run for each crafted and application instance.

4.3.2. Comparing Swcca with Swcc and Sparrow2011 on random 3-SAT

On 3-SAT Comp11 Benchmark: Table 3 presents the results of comparing Swcca with Swcc and Sparrow2011 on the 3-SAT Comp11 benchmark. Swcca shows a substantial improvement over Swcc on these random 3-SAT instances. On all instance classes except for the two easy ones (3SAT-v2500 and 3SAT-v5000), Swcca achieves a higher success rate than Swcc does, and the gap increases with the size of instances. Particularly, on the largest sized instances ($n = 50\,000$), Swcca succeeds in all runs while Swcc only succeeds in about half runs. Table 3 also indicates that Swcca significantly outperforms Sparrow2011 in terms of both success rate and run time. Specially, altogether there is only one instance on which Swcca does not achieve a 100% success rate, where it achieves a 99% success rate. However, this is also the most difficult instance for Sparrow2011 and its success rate is only 48%. The obvious gap of success rate between the two solvers on the largest size group ($n = 50\,000$) indicates a substantial performance improvement of Swcca over Sparrow2011 on these large random 3-SAT instances. The superiority of Swcca over Swcc and Sparrow2011 in terms of averaged run time is also clearly illustrated in Fig. 2.

On 3-SAT Huge Benchmark: Table 4 presents the comparative performance results between Swcca and Sparrow2011 on random 3-SAT instances which are larger than those in SAT competitions. The results demonstrate that Swcca significantly outperforms Sparrow2011 on these large random 3-SAT instances. The success rates of Swcca on all groups of these huge instances are 100% or almost that, while those of Sparrow2011 vary from 68% to 94.8%. In the respect of averaged run time, Swcca is 2–3 times faster than Sparrow2011. We also report the averaged number of flips on each instance class, to provide a view on the comparative performance of implementation independent between Swcca and Sparrow2011 on random 3-SAT instances. The results show that Swcca is 3–4 times faster than Sparrow2011 in terms of averaged flips. On the other hand, the observation that the flip number gap is bigger than the run time gap indicates that Swcca spends more time in each step than Sparrow2011 does, which might be due to the maintenance of CCD variables.

Table 5
Comparative performance results of Swcca, sattime and glucose on the crafted benchmark. The results in bold indicate the best performance for an instance class.

Instance class (#instance)	Sattime		Swcca		Glucose	
	suc rate	avg time	suc rate	avg time	suc rate	avg time
289 (15)	100%	< 0.01	100%	< 0.01	93.3%	121
automata-synchronization (7)	0%	n/a	17.5%	1498	**85.7%**	**422**
battleship (14)	98.6%	26	**100%**	**< 0.01**	50%	926
GreenTao (3)	**100%**	**37**	86.7%	91	33.3%	1277
sgen (10)	**96%**	**199**	35%	1041	20%	1461
SRHD-SGI (28)	**52.8%**	**930**	47.1%	1059	21.3%	1529
VanDerWaeden_pd_3k (7)	75.7%	646	**100%**	**134**	85.7%	853

Table 6
Comparative performance results of Swcca, sattime and glucose on the application benchmark. The results in bold indicate the best performance for an instance class.

Instance class (#instance)	Sattime		Swcca		Glucose	
	#solved	avg time	#solved	avg time	#solved	avg time
2dimensionalstrippacking (6)	0	n/a	0	n/a	6	32
AAAI2010-SATPlanning (6)	1	1751	1	1501	5	329
AES (13)	1	1662	1	1684	2	1523
AProVE (8)	0	n/a	1	1611	8	204
SATPlanning (17)	1	1784	1	1702	5	1399
slp-synthesis-AES (19)	0	n/a	0	n/a	0	n/a
smtqfbv-aigs (7)	1	1544	1	1556	5	519
traffic (2)	0	n/a	1	1105	2	28

4.3.3. Comparing Swcca with sattime and glucose on structured benchmarks

On Crafted Benchmark: The experimental results on the crafted benchmark are reported in Table 5, which illustrate that both SLS solvers Swcca and sattime are overall better than (or at least competitive with) the complete solver glucose on these crafted instances. When comparing the two SLS solvers, Swcca is competitive with and complementary to sattime, in the sense that among the 7 types of crafted instances in the benchmark, Swcca outperforms sattime on 3 types while sattime outperforms Swcca on 3 other types. Particularly, while sattime fails on all automata-synchronization instances, Swcca finds satisfying assignments for three of them. To the best of our knowledge, this is the first time that a local search SAT solver solves such automata-synchronization instances. Finally, we would like to note that sattime utilizes some reasonings before the local search procedure, which is helpful for solving crafted instances, while Swcca does not employ any pre-process for simplifying the formulas.

On Application Benchmark: Of the 78 application instances, glucose solves 33 of them, while Swcca and sattime solve 6 and 4 respectively, which demonstrates the significant performance gap between CDCL-based and SLS-based solvers on application instances. Nevertheless, when considering the number of instance classes where solvers have at least one successful run, Swcca solves an instance for 6 different instance classes out of 8, and this figure is 4 for sattime and 7 for glucose (see Table 6).

To summarize, both local search solvers perform much worse than the complete solver glucose on application instances, although they can solve crafted instances on a level competitive with (or better than) complete solvers. This is interesting as local search algorithms have proved successful for application instances in many optimization problems. For example, local search algorithms find optimal solutions much faster than complete algorithms for most instances in the DIMACS Challenge Maximum Clique benchmark, including the application ones [11]. Thus, a significant challenge for SAT local search algorithms is to improve their performance on application instances.

5. Subscore and the CCAsubscore algorithm

This section improves Swcca for random k-SAT with $k > 3$. First, we replace the SWT scheme in Swcca with PAWS [46], leading to a new algorithm called CCApaws. More importantly, we propose a new variable property called *subscore*, and utilize it to break ties in CCApaws, resulting in the algorithm CCAsubscore, which proves very efficient for random k-SAT with $k > 3$. Our experiments show that CCAsubscore significantly outperforms other state-of-the-art local search algorithms on random 5-SAT and 7-SAT instances.

5.1. The CCApaws algorithm

Although Swcca exhibits good performance on random 3-SAT and crafted instances, its performance on random k-SAT instances with large k is relatively weak. Our experiments suggest that the SWT weighting scheme is ineffective for random k-SAT instances with large k. As a first step towards improving Swcca for such instances, we replace the SWT scheme with

the PAWS one [46], which results in the CCApaws algorithm. Note that the PAWS scheme is adopted by state-of-the-art SLS algorithms for SAT such as Sparrow2011 [3] and EagleUP [15], which show better performance than Swcca on random k-SAT instances with $k > 3$.

The PAWS Scheme: whenever PAWS is called, the clause weights are updated as follows. With probability sp (smoothing probability), for each satisfied clause whose weight is larger than one, the clause weight is decreased by one. Otherwise, the clause weights of all falsified clauses are increased by one.

CCApaws is modified from Swcca, and the only difference between the two algorithms is the clause weighting scheme. According to our experiments, CCApaws performs considerably better than Swcca on random 5-SAT instances (Table 8), but it does not show any notable improvement on random 7-SAT instances (Table 9). On the other hand, CCApaws cannot rival state-of-the-art SLS solvers, such as the winners of SAT Competition 2011, on random 5-SAT and 7-SAT instances, and thus needs to be further improved.

5.2. Subscore

Local search algorithms view the SAT problem as an optimization problem whose goal is to minimize the number of falsified clauses. Some variable properties, such as *score* and *break*, are thus commonly used, which concern the state transformations of clauses by flipping a variable. In our opinion, however, differentiating clauses by their states is not informative enough to guide the local search in solving random k-SAT instances with large k, for the number of true literals varies considerably among satisfied clauses in such instances. We take a further step by distinguishing different kinds of satisfied clauses in terms of the number of true literals. A special type of satisfied clauses is those containing only one true literal. These clauses, although being satisfied, would become falsified more easily than other satisfied clauses. For convenience, we give the following definitions.

Definition 3. Given a CNF formula F and an assignment α to $V(F)$, a satisfied clause is *critical* if and only if it has only one true literal under assignment α; otherwise, it is a *stable clause*.

When we are talking about critical and stable clauses, the assignment α is usually explicit from the context, and thus omitted.

In some sense, the above definition of critical clauses is an extended version of a concept also named critical clauses proposed in the theory community [38], which is restricted to satisfying assignments (i.e., a clause is critical if there is exactly one true literal in that clause under a *satisfying* assignment). The concept of critical clauses is very important in the analyses of the exponential-time algorithm PPSZ [38], which is well-known as a milestone of improving upper bounds for random k-SAT problems.

The differentiation of critical clauses from stable ones here is also of great importance. We believe that apart from the number of falsified clauses, it would be beneficial for SLS algorithms to take into account the number of critical clauses when choosing a variable for flipping. For example, if more than one variable have the greatest score, then it is advisable to select the one whose flip leads to the minimum number of critical clauses.

Based on the above considerations, we propose a new variable property called *subscore*. When more than one candidate variable have the greatest score, those with greater subscore are more promising to improve the objective function and thus should be given higher priorities to be flipped. Formally, we have the following definitions.

Definition 4. For a variable x, its *submake* is the number of critical clauses that would become stable by flipping x. Similarly, its *subbreak* is the number of stable clauses that would become critical by flipping x. The *subscore* of x, denoted by $subscore(x)$, is defined as $submake(x)$–$subbreak(x)$.

When considering clause weights in dynamic local search, $submake(x)$ measures the total weight of the critical clauses that would become stable clauses by flipping x, and $subbreak(x)$ does that of the stable clauses that would become critical by flipping x.

5.3. The CCAsubscore algorithm

We utilize the subscore property to break ties in CCApaws, resulting in the CCAsubscore algorithm. The pseudo codes of CCAsubscore are shown in Algorithm 4.

CCAsubscore differs from CCApaws only in the tie-breaking mechanism in the greedy mode. In detail, CCApaws breaks ties by preferring the oldest variable, while CCAsubscore does so by preferring the variable with the greatest subscore (further ties are broken by preferring the oldest one). The modification seems relatively small at first glance. In effect, however, it has an essential impact on the algorithm. This is because in each greedy step of CCAsubscore, a considerable portion of candidate variables have the equally greatest score, and thus the tie-breaking mechanism is very important to the algorithm.

We have performed some experiments to figure out the number of candidate variables that have the greatest score in each greedy step of CCAsubscore. The experiments are performed with the random 5-SAT and 7-SAT instances from SAT

Algorithm 4: CCAsubscore

```
Input: CNF-formula F, maxSteps
Output: A satisfying assignment α of F, or "no solution found"
begin
    α := randomly generated truth assignment;
    for step := 1 to maxSteps do
        if α satisfies F then return α;
        if there exist decreasing variables whose configuration has been changed after its last flip then
            v := such a variable with the greatest score, breaking ties by preferring the one with the greatest subscore;
        else if there exist variables with a score higher than w̄ then
            v := such a variable with the greatest score, breaking ties by preferring the one with the greatest subscore;
        else
            update clause weights according to the PAWS scheme;
            v := the oldest variable in a random falsified clause;
        α := α with v flipped;
    return "no solution found";
end
```

Table 7
The averaged number and proportion of candidate variables that have the greatest score in each greedy step of CCAsubscore, based on 100 runs for each instance class.

	5SAT-v1500	5SAT-v2000	7SAT-v150	7SAT-v200
avg number	3.75	4.25	2.08	2.32
avg proportion	35%	31%	39%	32%

Competition 2011, and the results are summarized in Table 7. As is demonstrated in Table 7, on average, there are more than three candidate variables that have the greatest score for these random 5-SAT instances and more than two such variables for the random 7-SAT instances, in each greedy step of CCAsubscore. Moreover, these candidate variables with the greatest score occupy a considerable proportion (more than 30%) of all candidate variables. Therefore, the tie-breaking mechanism plays a substantial role in the CCAsubscore algorithm.

5.4. Evaluations of CCAsubscore

In this subsection, we carry out experiments to evaluate the performance of CCAsubscore on random 5-SAT and 7-SAT instances. First, we compare the three algorithms Swcca, CCApaws and CCAsubscore, to demonstrate the effectiveness of the PAWS scheme and the tie-breaking mechanism based on subscore in CCAsubscore. Then, we compare CCAsubscore against other state-of-the-art SLS solvers for SAT.

5.4.1. Benchmarks and experiment preliminaries

To evaluate CCAsubscore, we set up two benchmarks, one for random 5-SAT and the other for random 7-SAT.

1. **5-SAT benchmark:** all large random 5-SAT instances from SAT Competition 2011 ($r = 20$, $1000 \leqslant n \leqslant 2000$, 10 instances for each size).
2. **7-SAT benchmark:** large random 7-SAT instances from SAT Competition 2011 ($r = 85$, $150 \leqslant n \leqslant 250$, 10 instances for each size), and 200 randomly generated instances ($r = 85$, $220 \leqslant n \leqslant 240$, 100 instances for each size). Those 7-SAT instances with 300 and 400 variables from SAT Competition 2011 are too hard for all solvers so that they are not included in our experiments. Instead, we generated 200 instances with 220 and 240 variables to extend the benchmark.

Both CCApaws and CCAsubscore are implemented in C++ and compiled by g++ with the '-O2' option. As for the *sp* parameter in the PAWS scheme, some parameter-tuning experiments for CCApaws and CCAsubscore suggest that the optimal value of *sp* is between 0.7 to 0.74 for k-SAT with $k = 4, 5$, and is between 0.9 and 0.94 for k-SAT with $k = 6, 7$. When *sp* is in the corresponding optimal interval, the performance of CCAsubscore (and also CCApaws) varies very slightly. We set *sp* to 0.72 for k-SAT with $3 < k \leqslant 5$ and 0.92 for k-SAT with $k > 5$ for both CCApaws and CCAsubscore.

We compare the performance of CCAsubscore with four state-of-the-art SLS solvers, including the three top solvers from the random satisfiable category of SAT Competition 2011 (i.e., Sparrow2011, sattime2011 and EagleUP), as well as the recent SLS solver probSAT [5]. Note that probSAT represents the latest breakthrough, to our knowledge, in solving random k-SAT instances. Particularly, on the large random benchmark from SAT Competition 2011, the adaptive version of probSAT (setting parameters according to the size of the clauses) significantly outperforms the three winners of SAT Competition 2011 and WalkSAT with the adaptive parameter. For our experiments, we adopt the adaptive version of probSAT [5].

Table 8
Comparative performance results of Swcca, CCApaws and CCAsubscore on the 5-SAT benchmark. Each solver is performed 100 times on each instance class with a cutoff time of 5000 seconds.

Instance class	Swcca		CCApaws		CCAsubscore	
	suc rate	avg time	suc rate	avg time	suc rate	avg time
5SAT-v750	81%	2038	100%	235	100%	47
5SAT-v1000	23%	4071	100%	316	100%	81
5SAT-v1250	0	n/a	100%	434	100%	128
5SAT-v1500	0	n/a	71%	2585	100%	443
5SAT-v2000	0	n/a	30%	4171	93%	1586

Table 9
Comparative performance results of Swcca, CCApaws and CCAsubscore on the 7-SAT benchmark. Each solver is performed 100 times on each instance class with a cutoff time of 5000 seconds.

Instance class	Swcca		CCApaws		CCAsubscore	
	suc rate	avg time	suc rate	avg time	suc rate	avg time
7SAT-v150	68%	2488	71%	2591	100%	232
7SAT-v200	5%	4889	6%	4870	72%	2312
7SAT-v220	0	n/a	0	n/a	68%	2798
7SAT-v240	0	n/a	0	n/a	33%	4008
7SAT-v250	0	n/a	0	n/a	7%	4881

The solvers Sparrow2011, sattime2011, and EagleUP are downloaded online from the SAT competition website,[5] and the binary of probSAT is kindly provided by its author.

Experiments in this section are performed on the EDACC platform [4], running on 2 cores of 2.8 GHz Intel(R) Core(TM) i7-2640M CPU and 7.8 GB RAM of a work station under Linux. Each solver is performed 100 times for each instance class, with a cutoff time of 5000 seconds.

5.4.2. Comparing CCAsubscore with Swcca and CCApaws

In this subsection, we compare CCAsubscore with Swcca and CCApaws on both random 5-SAT and 7-SAT benchmarks.

On 5-SAT Benchmark: As is clear from Table 8, the performance of Swcca is much worse than those of the other two algorithms. Due to the replacement of the SWT scheme by PAWS, CCApaws gains a significant improvement over Swcca for random 5-SAT instances. Further, the utilization of subscore renders CCAsubscore much more efficient than CCApaws. In the respect of averaged run time, CCAsubscore is about four times faster than CCApaws. Moreover, on the largest sized 5SAT-v2000 instances, the success rate of CCAsubscore is more than three times higher than that of CCApaws.

On 7-SAT Benchmark: Table 9 shows that the performance of CCApaws is very similar to that of Swcca, which indicates that replacing the SWT scheme with the PAWS one does not lead to any obvious improvement for random 7-SAT instances. On the other hand, CCAsubscore shows a substantial improvement over CCApaws. In particular, while CCApaws fails to solve any instance with more than 200 variables, CCAsubscore still succeeds in 68% and 33% runs for the 7SAT-v220 and 7SAT-v240 instances respectively. Since the only difference between CCAsubscore and CCApaws is the tie-breaking mechanism, we attribute the good performance of CCAsubscore to its tie-breaking mechanism based on subscore.

We also observe that the performance of Swcca for random 5-SAT and 7-SAT instances cannot be obviously improved by tuning the parameters for SWT, which suggests that the SWT scheme is likely not suitable for solving such instances. We tried all parameter combinations of $\gamma = 100, 200, \ldots, 1000$ (the performance of Swcca degrades significantly when γ exceeds 1000) and $\rho = 0.1, 0.2, \ldots, 0.9$. None of the settings can solve any 5SAT-v1250 instance, or get a 100% success rate for the 7SAT-v150 instance class.

5.4.3. Comparing CCAsubscore with state-of-the-art SLS solvers

In this subsection, we compare CCAsubscore with state-of-the-art SLS solvers for SAT on the random 5-SAT and 7-SAT benchmarks.

On 5-SAT Benchmark: Table 10 shows the comparative results between CCAsubscore and other state-of-the-art SLS algorithms on the 5-SAT benchmark. We observe that the performance of Sparrow2011 and probSAT are very similar on these random 5-SAT instances, and are obviously better than those of sattime2011 and EagleUP. However, CCAsubscore shows a clear superiority over Sparrow2011 and probSAT on this benchmark. As is clear from the results, CCAsubscore significantly outperforms other solvers on all instance classes in terms of both success rate and averaged run time. Particularly, on the largest instances, i.e., the 5SAT-v2000 instances, the success rate of CCAsubscore is 93%, while this figure is 72% and 71% for Sparrow2011 and probSAT, 10% and 25% for sattime2011 and EagleUP, respectively. For the 5SAT-v1500 instances, CCAsub-

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

Table 10
Comparison of CCAsubscore with other state-of-the-art SLS solvers on the 5-SAT benchmark. Each cell summarizes the performance of the solver for 100 runs on each instance class with a cutoff time of 5000 seconds. The top row shows the success rate and the bottom row shows the averaged run time for each instance class.

	Sparrow2011 suc rate avg time	sattime2011 suc rate avg time	EagleUP suc rate avg time	probSAT suc rate avg time	CCAsubscore suc rate avg time
5SAT-v750	100%	99%	100%	100%	100%
	51	302	72	88	**47**
5SAT-v1000	100%	97%	100%	100%	100%
	159	2154	184	185	**81**
5SAT-v1250	100%	91%	100%	100%	100%
	174	1538	384	237	**128**
5SAT-v1500	99%	52%	88%	98%	**100%**
	781	3501	1823	853	**443**
5SAT-v2000	72%	10%	25%	71%	**93%**
	2688	4749	4281	2585	**1586**

Table 11
Comparison of CCAsubscore with other state-of-the-art SLS solvers on the 7-SAT benchmark. Each cell summarizes the performance of the solver for 100 runs on each instance class with a cutoff time of 5000 seconds. The top row shows the success rate and the bottom row shows the averaged run time for each instance class.

	Sparrow2011 suc rate avg time	sattime2011 suc rate avg time	EagleUP suc rate avg time	probSAT suc rate avg time	CCAsubscore suc rate avg time
7SAT-v150	100%	100%	98%	88%	100%
	642	512	445	1580	**232**
7SAT-v200	17%	46%	48%	11%	**72%**
	4562	3608	3625	4756	**2312**
7SAT-v220	13%	35%	34%	10%	**68%**
	4706	4100	4061	4723	**2798**
7SAT-v240	2%	12%	11%	2%	**33%**
	4920	4589	4782	4951	**4008**
7SAT-v250	0	0	2%	0	**7%**
	n/a	n/a	4976	n/a	**4881**

score is about two times faster than Sparrow2011 and probSAT, eight times faster than sattime, and four times faster than Eagle.

On 7-SAT Benchmark: The comparative results on the random 7-SAT benchmark are presented in Table 11. None of these solvers can solve the 7SAT-v200 instances consistently, which indicates that solving random 7-SAT instances near the phase transition is still a challenge for modern SAT solvers. On the other hand, CCAsubscore significantly outperforms other solvers on all these random 7-SAT instances, in terms of both averaged run time and success rate. Specially, on the 7SAT-v240 instances, CCAsubscore achieves a success rate of 33%, which is 15 times higher than those of Sparrow2011 and probSAT, and 3 times higher than those of sattime2011 and EagleUP.

To sum up, the experiments demonstrate that CCAsubscore consistently outperforms its competitors on both random 5-SAT and 7-SAT instances. These state-of-the-art solvers for comparison can be categorized into two groups. Sparrow2011 and probSAT show promising results on random 5-SAT instances, but their performance on random 7-SAT instances are much weaker than other solvers. In contrast, sattime2011 and EagleUP show superiority over Sparrow2011 and probSAT on random 7-SAT instances, but they lose their power when coming to large random 5-SAT instances. Encouragingly, CCAsubscore exhibits much better performance than all these four solvers on both benchmarks, indicating a remarkable progress in solving random k-SAT instances with $k > 3$.

Nevertheless, CCAsubscore performs poorly on random 3-SAT instances. Also, the notion of subscore does not improve Swcca on random 3-SAT instances. In fact, we will prove in Section 7.4 that the notion of subscore is not suitable for solving random 3-SAT instances.

6. The CCASat solver and results on random track of SAT Challenge 2012

As both Swcca and CCAsubscore are based on the CCA framework, we combine them and obtain a local search SAT solver called CCASat. Specifically, CCASat adopts Swcca to solve random 3-SAT and structured instances, and CCAsubscore to solve random k-SAT instances with $k > 3$. The CCASat solver is implemented in C++ and compiled by g++ with the '-O2' option. The parameters for Swcca and CCAsubscore are set the same as those in the experiments in Sections 4 and 5 respectively.

In this section, we present and analyze the experimental results of the random track in SAT Challenge 2012, where CCASat was ranked first.

Table 12
Threshold values r_k for different k in the random k-SAT model.

k	3	4	5	6	7
r_k	4.267	9.931	21.117	43.37	87.79

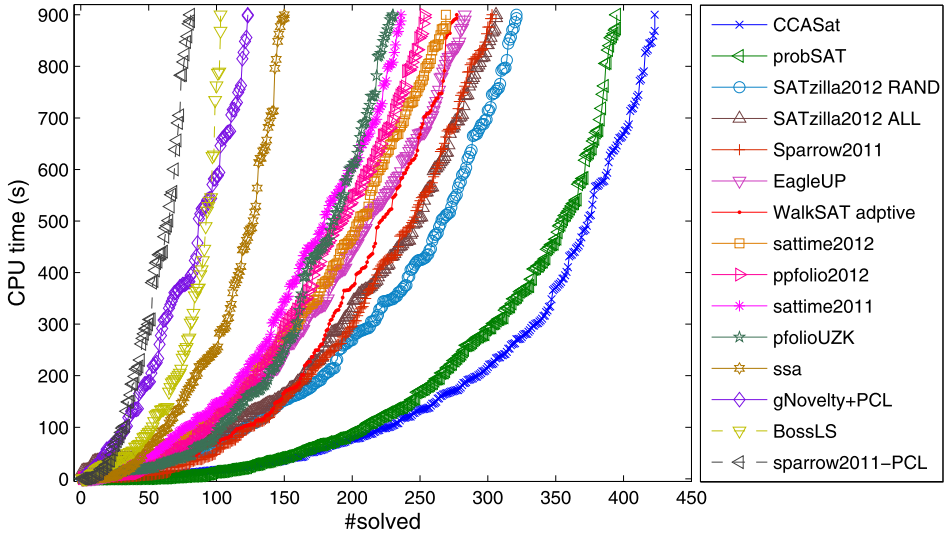

Fig. 3. Comparing CPU time distributions for SAT solvers in the random track of SAT Challenge 2012, where the cutoff time is 900 seconds. The probSAT and WalkSAT adaptive solvers are off track and are not considered in the official ranking online.

6.1. The SAT Challenge 2012 random benchmark

SAT Challenge 2012 is a competitive event for solvers of the Boolean Satisfiability (SAT) problem. It is organized as a satellite event to the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT 2012) and stands in the tradition of the SAT competitions that have been held yearly from 2002 to 2005 and biannually starting from 2007, and the SAT-Races held in 2006, 2008 and 2010.[6]

All instances from the random track of SAT Challenge 2012 (600 instances, 120 for each k-SAT, $k = 3, 4, 5, 6, 7$) are generated according to the random k-SAT model near the threshold ratios, which have been cited as the hardest group of SAT problems [24]. For large n, the best approximations of the threshold ratios are given in [33] and listed in Table 12.

Specifically, the instances are distributed from 2000 variables at $r = 4.267$ to 40 000 variables at $r = 4.2$ for 3-SAT, from 800 variables at $r = 9.931$ to 10 000 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.

6.2. Results of SAT Challenge 2012 random track

The data of experimental results on the random track of SAT Challenge 2012 is taken from the website for SAT Challenge 2012.[7] We present the CPU time distributions for all participating solvers in Fig. 3. As can be seen from Fig. 3, CCASat is the best solver among the participants, including those off track ones (probSAT and WalkSAT adaptive) which are submitted by the organizers and are not considered in the official ranking.

To demonstrate the detailed performance of CCASat on random k-SAT instances with various k ($k = 3, 4, 5, 6, 7$), we report in Table 13 the number of solved instances for each solver on each k-SAT. Table 13 shows that CCASat performs quite well on all types of random k-SAT instances with various k, and solves the most instances for the whole benchmark. Specially, CCASat solves the most instances for k-SAT with $k = 5, 6, 7$. On the random 4-SAT instances where CCASat is ranked second, the best solver probSAT solves only three more instances than CCASat does. Table 13 also reports for each solver the averaged run time along with standard deviation, and the median time over all runs, all of which confirm the significant superiority of CCASat over other solvers.

On the other hand, for random 3-SAT instances, the best-performing solvers belong to a special type of SLS algorithms called *focused random walk* [37], such as BossLS [14], probSAT [5], and WalkSAT [43]. These focused random walk algorithms

[6] http://baldur.iti.kit.edu/SAT-Challenge-2012.
[7] http://edacc2.informatik.uni-ulm.de/SATChallenge2012/experiment/19/property-distribution/.

Table 13
Ranking results on random track of SAT Challenge 2012, where cutoff time is 900 seconds. There are 120 k-SAT instances for each k. The solvers marked with * are winners of the random satisfiable category of SAT Competition 2011. The solvers marked with ** are submitted by the organizers of SAT Challenge 2012, and thus are not considered in the official ranking in SAT Challenge 2012. We report their results for comparison in our presentation.

Solver	#solved						avg time (std dev)	Median time over all
	3-SAT	4-SAT	5-SAT	6-SAT	7-SAT	ALL		
CCASat (this work)	76	112	**60**	**97**	**78**	**423**	**392** (376)	**218.8**
probSAT** [5]	90	**115**	53	78	59	395	423 (383)	292.9
SATzilla2012 RAND [50]	61	63	46	79	72	321	533 (366)	714.4
SATzilla2012 ALL [50]	60	75	48	58	65	306	579 (360)	845.6
Sparrow2011* [3]	55	74	52	59	63	303	572 (372)	876.1
EagleUP* [15]	52	42	39	80	70	283	615 (349)	900.0
WalkSAT adaptive** [5]	80	63	22	62	50	277	597 (370)	900.0
sattime2012 [29]	44	42	35	85	63	269	630 (346)	900.0
ppfolio2012 [40]	36	84	44	35	54	253	638 (350)	900.0
sattime2011* [27]	26	46	23	69	72	236	658 (340)	900.0
pfolioUZK [49]	36	81	30	33	50	230	647 (358)	900.0
ssa [45]	58	27	24	0	41	150	733 (313)	900.0
gNovelty + PCL [39]	4	13	15	38	53	123	782 (257)	900.0
BossLS [14]	**98**	0	5	0	0	103	777 (283)	900.0
sparrow2011-PCL [39]	41	0	0	1	39	81	816 (233)	900.0

focus the search by always selecting a variable to flip from a falsified clause (chosen at random). Indeed, focused random walk approaches have proved very successful on random 3-SAT instances, especially those with $r = 4.2$ [42,25].

Although CCASat cannot overall rival the focused random walk algorithms on random 3-SAT instances, it gives the best performance on those 3-SAT instances whose ratios are very close to the phase transition. On the 24 random 3-SAT instances with $r = 4.26, 4.267$, only 3 of them CCASat fails to solve within the cutoff time, while this number is 10 for BossLS, 9 for probSAT and 13 for WalkSAT. Apart from the focused random walk algorithms, other solvers also solve fewer such random 3-SAT instances than CCASat does. This is remarkable as solving the random 3-SAT instances very close to the phase transition is a big challenge for all kinds of algorithms. A recent local search solver called FrwCB can solve random 3-SAT instances at $r = 4.2$ with up to 4 million variables within reasonable time, but its performance on those with ratio 4.267 is worse than that of CCASat [30]. The Survey Propagation (SP) algorithm [7], which is extremely good at solving random 3-SAT instances, can solve random 3-SAT instances at $r = 4.2$ with up to 10 million variables [35], but it fails to converge on those with ratio up to 4.267, or before that [7].

7. Further analyses

In this section, we conduct further analyses to provide more insights into the configuration checking strategy and the subscore property. Specifically, we reveal the relationships between the CC strategy and other two related heuristics, analyze the impact of the parameters of instance on the effectiveness of the CC strategy, discuss the implementations of the CC strategy, and show that the subscore property is not suitable for solving random 3-SAT instances.

7.1. Comparison of configuration checking with related heuristics

We demonstrate the relationship between CCD variables and promising decreasing variables, as well as the relationship between the CC strategy and the tabu method, in terms of forbidding strength.

7.1.1. CCD variables vs. promising decreasing variables

In the following, we discuss the relationship between CCD variables and promising decreasing variables [26], which are widely used in state-of-the-art SLS solvers for SAT. Proposed in G^2WSAT [26], the concept of promising decreasing variables has been widely used to improve the greedy mode of SLS algorithms for SAT. All awarded SLS solvers in SAT competitions 2007, 2009 and 2011 follow G^2WSAT and switch between the greedy and diversification modes depending on the existence or not of promising decreasing variables. However, the CCA heuristic switches between the two modes depending on the existence of the CCD and SD variables.

We consider the unweighted version of CCD variables and promising decreasing variables, where $score(x)$ is the difference between the number of falsified clauses under the current assignment α and that under the assignment obtained by flipping x in α. Nevertheless, the conclusions can be easily extended to the weighted version.

First, we would like to recall some concepts:

- A variable x is *decreasing* iff $score(x) > 0$, and *increasing* iff $score(x) < 0$.
- A *configuration changed decreasing* (CCD) variable is a decreasing variable that $confChange(x) = 1$.

- [26] Let x be a variable which is not decreasing. If it becomes decreasing after another variable y is flipped, then we say that x is a *promising decreasing variable* after y is flipped. For a promising decreasing variable x, it remains promising as long as it is decreasing after one or more other flips.

For the relationship between CCD variables and promising decreasing variables, we have the following conclusions.

Proposition 5. *For a given variable x, if x is a promising decreasing variable, then x is a CCD variable.*

Proof. The proof is given by induction.

(a) *Becoming a promising decreasing variable.* If x becomes a promising decreasing variable after flipping another variable y, then we conclude $y \in N(x)$. Otherwise, y is independent of x and flipping y does nothing to $score(x)$. Since $y \in N(x)$, along with flipping y, $confChange(x)$ would be set to 1. As x is a decreasing variable and $confChange(x) = 1$, x is a *CCD* variable by definition.

(b) *Remaining a promising decreasing variable.* For a promising decreasing variable x, if x remains promising decreasing, then x has not been flipped after the last time it became a promising decreasing variable. Otherwise, because x is decreasing, i.e., $score(x) > 0$, flipping x would make $score(x) < 0$ (flipping x would make $score(x)$ be its opposite number). But this means x is no longer a decreasing variable, and thus not a promising decreasing variable. Recalling that only flipping x can set $confChange(x)$ to 0, we conclude that $confChange(x)$ remains 1. Thus, x remains a *CCD* variable. □

Remark 6. The reverse of Proposition 5 is not necessarily true.

Proof. For a variable x to be CCD, it suffices that one of its neighboring variable is flipped and $score(x) > 0$. To be promising, one or several neighboring variables should be flipped to make its score positive. When an increasing variable is flipped, it is CCD as soon as one of its neighboring variables is flipped and its score remains positive. However, it cannot be promising until its neighboring variables are flipped to make its score non-positive and then positive. □

To sum up, our analysis shows that promising decreasing variables are a subset of CCD variables. This indicates if a variable is forbidden by the CC strategy, it is also forbidden by the promising decreasing strategy, but the reverse is not necessarily true. Therefore, the forbidding strength of CC is not so strong as that of the promising decreasing strategy. In some sense, CCD variables and promising decreasing variables may be two extremities, and there may be an intermediate notion more effective to be investigated in the future.

7.1.2. Configuration checking vs. tabu

As we have mentioned before, the tabu method is a previous significant method for handling the cycling problem in local search, and has been widely used in local search algorithms. Recalling that the tabu method for SAT forbids flipping a variable whose *age* is less than the tabu tenure (tt), we have the following conclusions for the relationship between the configuration checking strategy and the tabu method.

Proposition 7. *For a given variable x, if x is forbidden to flip by the tabu method with $tt = 1$, then $confChange(x) = 0$.*

Proof. If at the current search step, x is forbidden to flip by the tabu mechanism with $tt = 1$, then x is the variable that just be flipped at last step ($confChange(x)$ would be set to be 0). Since there is no flip between last step and the current step, $confChange(x)$ would be still 0 when selecting the variable to flip at the current step. □

Remark 8. The reverse of Proposition 7 is not necessarily true.

According to Proposition 7 and Remark 8 we conclude that the forbidding strength of the CC strategy is stronger than that of the tabu mechanism with $tt = 1$. Actually, in the next subsection, we will show the case in which the CC strategy degrades to the tabu mechanism with $tt = 1$.

7.2. Size of neighborhood and effectiveness of configuration checking

In this subsection, we illustrate the impact of the parameters of a formula such as k (the clause length), n (the number of variables) and r (the clause-to-variable ratio) on the effectiveness of the CC strategy, through both theoretical and experimental analysis.

Intuitively, if a SAT instance is so dense that most variables are connected to each other, then the configuration checking strategy becomes ineffective, because each flip causes most variables' *confChange* values to be 1. We will go deep into this intuition by analyzing the size of neighborhood of each variable in random k-SAT formulas, and demonstrating their influence on the effectiveness of the CC strategy.

For a random k-SAT formula $F_k(n, m)$, we calculate the size of each variable's neighborhood in F as following.

We fix an arbitrary variable x. $N(x)$ denotes the neighborhood of x. For any other variable y, $y \notin N(x)$ happens if and only if there does not exist such a clause that x and y both appear in.

For any clause, the probability that x and y both appear in the clause is

$$p = \frac{\binom{n-2}{k-2}}{\binom{n}{k}} = \frac{k(k-1)}{n(n-1)}$$

The probability that $y \in N(x)$ is

$$Pr\big(y \in N(x)\big) = 1 - Pr\big(y \notin N(x)\big) = 1 - (1-p)^m$$

Let I_y be the indicator variable for the event $\{y \in N(x)\}$, i.e.,

$$I_y = \begin{cases} 1, & \text{if } y \in N(x) \\ 0, & \text{if } y \notin N(x) \end{cases}$$

Then $\mathbb{E}(I_y) = Pr(y \in N(x)) = 1 - (1-p)^m$.

The expectation of the size of $N(x)$ can be obtained as following

$$\mathbb{E}\big(\#N(x)\big) = \mathbb{E}\Big(\sum_{y \in V(F)\setminus\{x\}} I_y\Big) \tag{1}$$

$$= \sum_{y \in V(F)\setminus\{x\}} \mathbb{E}(I_y) \tag{2}$$

$$= (n-1)\big(1 - (1-p)^m\big) \tag{3}$$

$$\approx (n-1)\left(1 - \left(\frac{1}{e}\right)^{pm}\right) \tag{4}$$

$$= (n-1)\left(1 - \left(\frac{1}{e}\right)^{\frac{k(k-1)}{n(n-1)}m}\right) \tag{5}$$

$$= (n-1)\left(1 - \left(\frac{1}{e}\right)^{\frac{k(k-1)r}{(n-1)}}\right) \tag{6}$$

We calculate the expectation of the size of a variable's neighborhood for the random k-SAT formulas near the threshold ratios, according to

$$\mathbb{E}\big(\#N(x)\big) \approx (n-1)\left(1 - \left(\frac{1}{e}\right)^{\frac{k(k-1)r}{(n-1)}}\right)$$

where r is the ratio of the formula. These expectations are listed in Table 14. We performed some experimental statistics to verify these theoretical expectations, and the experimental results are also reported in Table 14. The experimental results are surprisingly consistent with the theoretical ones.

As is clear from Table 14, random 3-SAT instances are so sparse that the size of each variable's neighborhood is very small. Actually, with the increase of the size of the formula, the size of each variable's neighborhood seems to be a constant near 25. We have already demonstrated the effectiveness of the CC strategy for random 3-SAT instances by comparing Swcc with its two alternatives (one just removes the CC strategy while the other replaces CC with the tabu method) in Section 3, which shows that CC is very effective and is better than the tabu method for random 3-SAT instances.

In the following, we demonstrate the effectiveness of CC on random 4-SAT and 5-SAT instances in the same way, by comparing CCAsubscore with its two alternatives, Gsubscore and TABUsubscore. These two alternative algorithms are obtained from CCAsubscore by modifying the greedy mode as below.

- Gsubscore: in the greedy mode, Gsubscore picks the decreasing variable with the greatest score, breaking ties by preferring the one with the greatest subscore.
- TABUsubscore (with tabu tenure tt): in the greedy mode, if there exists any decreasing variable whose age is greater than tt, TABUsubscore picks such a variable with the greatest score; otherwise, TABUsubscore picks an SD variable with the greatest score if any. All ties are broken by preferring the one with the greatest subscore, as in CCAsubscore.

For random 4-SAT instances near the phase transition, the size of each variable's neighborhood is also very small. We have carried out experiments to compare CCAsubscore with its two alternatives on the largest sized 4-SAT instances ($r = 9.0$) from SAT Challenge 2012, with the hardware specified in Section 5 and a cutoff time of 900 seconds. As what happens on random 3-SAT instances, CCAsubscore significantly outperforms its two alternatives on random 4-SAT instances. CCAsubscore

Table 14
The expectation of the size of each variable's neighborhood in a random k-SAT formula near the threshold ratio for $k = 3, 4, 5, 6, 7$. The first row shows the theoretical expectation while the second row shows the experimental size of each variable's neighborhood averaged over all variables on 50 instances for each instance class.

	3-SAT $r = 4.2$	4-SAT $r = 9.0$	5-SAT $r = 20$	6-SAT $r = 40$	7-SAT $r = 85$
$n = 100$	22.2	65.7	97.3	99.0	99.0
	(22.2)	(65.7)	(97.4)	(99.0)	(99.0)
$n = 200$	23.7	83.3	172.3	198.5	199.0
	(23.7)	(83.4)	(172.6)	(198.5)	(199.0)
$n = 500$	24.6	97.1	275.1	453.9	498.6
	(24.6)	(97.1)	(275.3)	(454.0)	(498.6)
$n = 1000$	24.9	102.4	329.6	698.5	971.0
	(24.9)	(102.4)	(329.6)	(698.3)	(971.0)
$n = 2000$	25.0	105.1	362.5	902.3	1663.9
	(25.0)	(105.1)	(362.5)	(902.2)	(1663.7)
$n = 5000$	25.1	106.8	384.4	1066.8	2551.4
	(25.1)	(106.8)	(384.4)	(1066.9)	(2551.4)
$n = 10\,000$	25.2	107.4	392.1	1130.8	3002.2
	(25.2)	(107.4)	(392.1)	(1130.8)	(3002.2)

Table 15
Comparison of CCAsubscore and its two alternatives Gsubscore and TABUsubscore on the 5-SAT benchmark. Each solver is performed 100 times on each instance class with a cutoff time of 5000 seconds.

Instance class	Size of neighborhood	Gsubscore		TABUsubscore			CCAsubscore	
		suc rate	avg time	opt tt	suc rate	avg time	suc rate	avg time
5SAT-v750	310 (0.41n)	100%	43	1	100%	**32**	100%	47
5SAT-v1000	330 (0.33n)	100%	79	1	100%	**72**	100%	81
5SAT-v1250	342 (0.27n)	100%	174	2	100%	**125**	100%	128
5SAT-v1500	350 (0.23n)	100%	548	4	100%	524	100%	**443**
5SAT-v2000	363 (0.18n)	61%	3170	5	88%	1880	**93%**	**1586**

is ten times faster than Gsubscore, and four times faster than TABUsubscore (with optimal tt). The results on random 4-SAT instances are not reported in detail. Comparatively, we focus on the performance comparisons of these three algorithms on random 5-SAT instances, where the results are more interesting and stimulating, as we can see there both situations where the CC strategy is effective and ineffective.

The comparative results of CCAsubscore and its alternatives Gsubscore and TABUsubscore on the 5-SAT benchmark are presented in Table 15. The comparison between CCAsubscore and Gsubscore shows that CCAsubscore improves Gsubscore on the 5-SAT instances with $n \geqslant 1250$, where the size of each variable's neighborhood is smaller than $0.3n$, while it performs similarly with Gsubscore on those instances with $n \leqslant 1000$, where the size of each variable's neighborhood is at least $0.3n$. We conjecture that the CC strategy becomes ineffective when the size of each variable's neighborhood is more than $0.3n$.

The comparison between CCAsubscore and TABUsubscore in Table 15 illustrates that the CC strategy and the tabu method (both with the aspiration mechanism) have their superiority in different situations. For the 5-SAT instances with $n \leqslant 1250$, where the size of each variable's neighborhood is more than $0.25n$, CCAsubscore performs worse than TABUsubscore. For those instances with $n \geqslant 1500$, where the size of each variable's neighborhood is less than $0.25n$, conversely, CCAsubscore outperforms TABUsubscore. Based on this observation, we conjecture that for SAT local search algorithms, if each variable's neighborhood is less than $0.25n$, it is better to utilize the CC strategy than the tabu method, and otherwise the tabu method is of more benefit. Also, we note that in order to achieve optimal performance, an algorithm using the tabu method needs to tune the tt parameter, or to design a reactive mechanism to eliminate the tt parameter, whereas the CC strategy does not introduce any parameter into algorithms.

Compared to random k-SAT with $k < 6$, random 6-SAT and 7-SAT instances (near the phase transitions) are much denser. As indicated in Table 14, for random 6-SAT instances with $n < 1500$ and 7-SAT instances with $n < 5000$, each variable's neighborhood consists of more than half of the variables in the formula. We will not be surprised that the CC strategy becomes trivial and ineffective on these instances. Indeed, we have carried out experiments on random 6-SAT instances from SAT Challenge 2012 and random 7-SAT instances from SAT Competition 2011, and there is not any observable performance difference among the three algorithms CCAsubscore, Gsubscore, and TABUsubscore. One possible explanation for this is that random 6-SAT and 7-SAT instances near the phase transitions are so difficult even when they are of relatively small size, and the cycling problem is not a main obstacle that hinders the performance of local search algorithms on them, until some breakthrough is made to enable us to solve such instances of larger size.

In the following, we show that if each variable's neighborhood is too large, the CC strategy degrades to the tabu method with tabu tenure 1, which forbids to flip the variable that was just flipped in last step. Recall that the CC strategy forbids a variable to be flipped if since its last flip, none of its neighboring variables has been flipped. For a given formula, it is

Table 16
The n^* value such that when $n < n^*$, the size of any variable's neighborhood is bigger than $n-2$. In these cases, the CC strategy degrades to the tabu method with tabu tenure 1.

Formulas	3-SAT ($r = 4.2$)	4-SAT ($r = 9.0$)	5-SAT ($r = 20$)	6-SAT ($r = 40$)	7-SAT ($r = 85$)
n^*	11.652	32.348	90.093	223.095	564.595

clear that if any two variables are neighboring to each other, then the CC strategy degrades to the tabu method with a tabu tenure of 1. The following proposition states the condition for this to happen.

Proposition 9. *For a random k-SAT formula $F_k(n, m)$, if $\ln(n-1) < \frac{k(k-1)m}{n(n-1)}$, the expected size of the neighborhood of any variable in F is larger than $n-2$.*

Proof. As we have established before, for any variable x in F, the expectation of the size of $N(x)$ is $\mathbb{E}(\#N(x)) = (n-1)(1-(1-p)^m)$.

Now, by using the inequality $(1-\frac{1}{n})^n < \frac{1}{e}$ $(n > 1)$, we have

$$\mathbb{E}\big(\#N(x)\big) = (n-1)\big(1-(1-p)^m\big) \tag{7}$$

$$> (n-1)\left(1-\left(\frac{1}{e}\right)^{pm}\right) \tag{8}$$

$$= (n-1)\left(1-\left(\frac{1}{e}\right)^{\frac{k(k-1)m}{n(n-1)}}\right) \tag{9}$$

$$= (n-1) - \frac{n-1}{e^{\frac{k(k-1)m}{n(n-1)}}} \tag{10}$$

When $n-1 < e^{\frac{k(k-1)m}{n(n-1)}}$, or equivalently, $\ln(n-1) < \frac{k(k-1)m}{n(n-1)}$, we have $\mathbb{E}(\#N(x)) > (n-1)-1 = n-2$. □

For a given random k-SAT formula $F_k(n, m)$, according to the above proposition, if $\ln(n-1) < \frac{k(k-1)r}{n-1}$ $(r = \frac{m}{n})$, then each variable is expected to have all other variables as its neighborhood. In this case, the CC strategy degrades to the tabu method with tabu tenure 1. Let $f(n) = \ln(n-1) - \frac{k(k-1)r}{n-1}$, it is clear that $f(n)$ is a monotonic increasing function of n $(n > 1)$. Thus, $f(n) < 0$ occurs if and only if $n \leqslant \lfloor n^* \rfloor$, where n^* is a real number such that $f(n^*) = 0$. We have calculated the value of such n^* for random k-SAT formulas $(k = 3, 4, 5, 6, 7)$ near the phase transition, as listed in Table 16.

We specially remark that the CC strategy degrades to the tabu method with $tt = 1$ for random 7-SAT instances at $r = 85$ with $n < 564$. On the other hand, modern local search solvers for SAT cannot solve such random 7-SAT instances with more than 300 variables (within reasonable time). This suggests that the CC strategy is currently not helpful for local search solvers in solving random 7-SAT instances, as we have mentioned before.

7.3. Approximate implementation of configuration checking

In this subsection, we discuss two different implementations for the CC strategy, including the approximate one in this paper and an accurate one. The complexity comparison between the two implementations shows that the approximate one has lower complexity and can be executed much more efficiently.

The implementation of the CC strategy in this paper (Section 3.3) is an approximate one and does not reflect its spirit accurately, recalling that the spirit of the CC strategy is to forbid flipping a variable x if none of its neighboring variables has changed its truth value since x was last flipped.

To see this, consider a variable x is flipped ($confChange(x)$ is set to 0); after that, another variable $y \in N(x)$ is flipped twice, which does not change y's truth value, but still makes $confChange(x)$ become 1 in our implementation. Suppose other neighboring variables of x have not changed their truth values during this period of time. In this case, the configuration for x is considered changed ($confChange(x) = 1$) according to our implementation, but it is not really changed since the true value of all x's neighbors are the same as the last time when x was flipped.

A naive accurate implementation of the CC strategy is to store the configuration (i.e., truth values of all its neighbors) for a variable x when it is flipped, and check the configuration when needed, say, when considering flipping x again.

Nevertheless, it is rather time-consuming to execute the CC strategy in this accurate way. For a formula F, we denote $\Delta(F) = max\{\#N(x)\colon\ x \in V(F)\}$. In this naive implementation, the CC strategy needs $O(\Delta(F))$ for both storing and checking the configuration for a variable. Therefore, the worst complexity per step for the CC strategy under the naive implementation (which needs to check the configuration for the candidate variables and store the configuration for the flipped variable) is $O(\Delta(F)n) + O(\Delta(F)) = O(\Delta(F)n)$.

For the approximate implementation in this work, in each step, the algorithm first scans the CCDVar stack to remove those variables that are no longer CCD; then it updates the *confChange* values for the flipping variable x and each $y \in N(x)$, and also adds those variables that become CCD from among $N(x)$ into the CCDVar stack. Therefore, the time complexity per step for the CC strategy is only $O(\#CCDVar) + 1 + O(\Delta(F))$, which is at most (usually much less) $O(n) + O(\Delta(F)) = O(n)$.

As is usual in local search algorithms, there is a trade-off between the accuracy of heuristics and the complexity per step. To lower the complexity of the CC strategy and thus the algorithms, we adopt the approximate implementation.

7.4. Further analyses on subscore

In this subsection, we show through theoretical analyses that the subscore property is not suitable for solving random 3-SAT instances.

The intuition is that for a given uniform random k-SAT, the number of true literals under any complete assignment is fixed (i.e., half of all literals). In particular, for any random 3-SAT formula, if too many clauses have more than one true literal under an assignment, then some clauses would have no true literal at all, and thus the corresponding assignment is not a model for the formula.

Proposition 10. *Given a uniform random k-SAT formula $F_k(n, m)$, there are at least $(2 - \frac{k}{2})m$ critical clauses under any satisfying complete assignment.*

Proof. Given a uniform random k-SAT formula $F_k(n, m)$, we denote the number of the clauses that have exactly i true literals ($0 \leqslant i \leqslant k$) as m_i. We are going to calculate a lower bound of m_1, i.e., the number of critical clauses, under any satisfying complete assignment. Let us consider an arbitrary satisfying complete assignment α, under which $m_0 = 0$. The number of true literals in F under α is denoted as $T(\alpha)$.

There are totally mk literals in $F_k(n, m)$, half of which are positive literals and the others are negative literals (when n approaches to $+\infty$, this is true with probability almost 1).

$$T(\alpha) = \frac{mk}{2} \tag{11}$$

Now we calculate $T(\alpha)$ in another way, by adding up true literals in the clauses with exactly i true literals for all values of i ($0 \leqslant i \leqslant k$).

$$T(\alpha) = \sum_{i=0}^{k} i m_i = \sum_{i=1}^{k} i m_i \tag{12}$$

$$= m_1 + \sum_{i=2}^{k} i m_i \tag{13}$$

$$\geqslant m_1 + 2\sum_{i=2}^{k} m_i \tag{14}$$

$$= m_1 + 2(m - m_1) \tag{15}$$

$$= 2m - m_1 \tag{16}$$

Eq. (15) holds because $m = m_0 + m_1 + \sum_{i=2}^{k} m_i = m_1 + \sum_{i=2}^{k} m_i$.

Putting Eq. (11) and inequation (16) together, we have

$$\frac{mk}{2} \geqslant 2m - m_1 \tag{17}$$

which yields a lower bound of the number of critical clauses as $m_1 \geqslant (2 - \frac{k}{2})m$. □

According to the above proposition, in order to make a random 3-SAT formula be evaluated true, at least half of the clauses ($(2 - \frac{3}{2})m = \frac{m}{2}$) should be critical, i.e., having only one true literal. Thus, the *subscore* property would lose its power on random 3-SAT formulas, since it encourages transformations from critical clauses to stable ones.

8. Conclusions and future work

This work proposed and analyzed two efficient local search heuristics for SAT, namely configuration checking (CC) and subscore, which break fresh ground from traditional SAT local search techniques. These novel heuristics have led to three efficient SLS algorithms for SAT, namely Swcc, Swcca and CCAsubscore. Swcc is our first endeavor to apply the idea of

configuration checking (CC) to local search for SAT. Swcca enhances Swcc by combining the CC strategy with an aspiration mechanism, while CCAsubscore improves Swcca by utilizing the notion of subscore.

Our experiments on random 3-SAT instances indicate that Swcc outperforms TNM, and Swcca does Sparrow2011. Note that TNM and Sparrow2011 wins the random satisfiable category of SAT Competition 2009 and 2011 respectively. Moreover, CCAsubscore significantly outperforms existing local search algorithms according to our experiments on large random 5-SAT and 7-SAT instances. Notably, the SAT solver CCASat which combines Swcca and CCAsubscore, was ranked the first in the random track of SAT Challenge 2012.

The CC strategy for SAT remembers each variable's configuration (truth values of all its neighboring variables), and forbids a variable to be flipped if its configuration has not been changed since its last flip. The CC strategy is further enhanced by an aspiration mechanism, resulting in a new *pickVar* heuristic called configuration checking with aspiration (CCA). According to CCA, there are two levels with different priorities in the greedy mode. Those variables whose flips can bring a big benefit have a chance to be selected on the second level, even if they do not satisfy the CC criterion.

The subscore property takes into account the transformations between critical clauses and steady clauses. By utilizing subscore to break ties in the CCA framework, CCAsubscore achieves much better performance on random k-SAT instances with $k > 3$ than existing SLS solvers do. The notion of subscore turns out very promising towards improving local search algorithms for random k-SAT instances with $k > 3$. This is remarkable, as in the past two decades, very few progress has been made in this direction.

Finally, we performed further analyses to gain deeper understandings of the CC strategy and the subscore property. We reveal the relationships of the CC strategy with the promising decreasing variable exploiting strategy and the tabu method, in terms of forbidding strength. Moreover, we give insights about the size of each variable's neighborhood in random k-SAT instances and its impact on the effectiveness of the CC strategy. We also discuss the complexity of two different implementations for the CC strategy, including the approximate one in this paper and an accurate one. For the subscore property, we show that it is unsuitable for solving random 3-SAT instances.

As for future work, we would like to design more sophisticated variable property and scoring functions related to subscore, which we believe can further improve the state of the art in SLS algorithms for SAT, especially for random k-SAT instances with $k > 3$. Given the promising results of Swcca on crafted instances, we also believe the CC heuristic can be helpful for solving such instances and we would like to go further in this direction by combining CC with other algorithmic techniques. Also we would like to apply the CC heuristic to other combinatorial search problems.

Acknowledgements

This work is supported by 973 Program 2010CB328103, ARC Grant FT0991785, National Natural Science Foundation of China (61073033, 61003056, 61272415, and 61370072), and the Fundamental Research Funds for the Central Universities of China grant 21612414.

We would like to thank the anonymous reviewers for their helpful comments on earlier versions of this paper.

References

[1] Dimitris Achlioptas, Random satisfiability, in: Handbook of Satisfiability, IOS Press, 2009, pp. 245–270.
[2] Gilles Audemard, Laurent Simon, Predicting learnt clauses quality in modern SAT solvers, in: Proc. of IJCAI-09, 2009, pp. 399–404.
[3] Adrian Balint, Andreas Fröhlich, Improving stochastic local search for SAT with a new probability distribution, in: Proc. of SAT-10, 2010, pp. 10–15.
[4] Adrian Balint, Daniel Gall, Gregor Kapler, Robert Retz, Experiment design and administration for computer clusters for SAT-solvers (edacc), JSAT 7 (2–3) (2010) 77–82.
[5] Adrian Balint, Uwe Schöning, Choosing probability distributions for stochastic local search and the role of make versus break, in: Proc. of SAT-12, 2012, pp. 16–29.
[6] Roberto Battiti, Marco Protasi, Reactive local search for the maximum clique problem, Algorithmica 29 (4) (2001) 610–637.
[7] Alfredo Braunstein, Marc Mézard, Riccardo Zecchina, Survey propagation: An algorithm for satisfiability, Random Struct. Algorithms 27 (2) (2005) 201–226.
[8] Shaowei Cai, Kaile Su, Local search with configuration checking for SAT, in: Proc. of ICTAI-11, 2011, pp. 59–66.
[9] Shaowei Cai, Kaile Su, Configuration checking with aspiration in local search for SAT, in: Proc. of AAAI-12, 2012, pp. 434–440.
[10] Shaowei Cai, Kaile Su, Qingliang Chen, EWLS: A new local search for minimum vertex cover, in: Proc. of AAAI-10, 2010, pp. 45–50.
[11] Shaowei Cai, Kaile Su, Chuan Luo, Abdul Sattar, NuMVC: An efficient local search algorithm for minimum vertex cover, J. Artif. Intell. Res. 46 (2013) 687–716.
[12] Shaowei Cai, Kaile Su, Abdul Sattar, Local search with edge weighting and configuration checking heuristics for minimum vertex cover, Artif. Intell. 175 (9–10) (2011) 1672–1696.
[13] Niklas Eén, Armin Biere, Effective preprocessing in SAT through variable and clause elimination, in: Proc. of SAT-05, 2005, pp. 61–75.
[14] Oliver Gableske, BossLS preprocessing and stochastic local search, in: Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions, 2012, pp. 10–11.
[15] Oliver Gableske, Marijn Heule, EagleUP: Solving random 3-SAT using SLS with unit propagation, in: Proc. of SAT-11, 2011, pp. 367–368.
[16] Luca Di Gaspero, Andrea Schaerf, A composite-neighborhood tabu search approach to the traveling tournament problem, J. Heuristics 13 (2) (2007) 189–207.
[17] Ian P. Gent, Toby Walsh, Towards an understanding of hill-climbing procedures for SAT, in: Proc. of AAAI-93, 1993, pp. 28–33.
[18] Fred Glover, Tabu search – part i, ORSA J. Comput. 1 (3) (1989) 190–206.
[19] Fred Glover, Tabu search – part ii, ORSA J. Comput. 2 (1) (1990) 4–32.
[20] Holger H. Hoos, Thomas Stützle, Local search algorithms for SAT: An empirical evaluation, J. Autom. Reason. 24 (4) (2000) 421–481.
[21] Holger H. Hoos, Thomas Stützle, Stochastic Local Search: Foundations & Applications, Elsevier/Morgan Kaufmann, 2005.

[22] Frank Hutter, Dave A.D. Tompkins, Holger H. Hoos, Scaling and probabilistic smoothing: Efficient dynamic local search for SAT, in: Proc. of CP-02, 2002, pp. 233–248.
[23] Henry A. Kautz, Ashish Sabharwal, Bart Selman, Incomplete algorithms, in: Handbook of Satisfiability, IOS Press, 2009, pp. 185–203.
[24] Scott Kirkpatrick, Bart Selman, Critical behavior in the satisfiability of random boolean formulae, Science 264 (1994) 1297–1301.
[25] Lukas Kroc, Ashish Sabharwal, Bart Selman, An empirical study of optimal noise and runtime distributions in local search, in: Proc. of SAT-10, 2010, pp. 346–351.
[26] Chu Min Li, Wen Qi Huang, Diversification and determinism in local search for satisfiability, in: Proc. of SAT-05, 2005, pp. 158–172.
[27] Chu Min Li, Yu Li, Satisfying versus falsifying in local search for satisfiability – (poster presentation), in: Proc. of SAT-12, 2012, pp. 477–478.
[28] Chu Min Li, Wanxia Wei, Yu Li, Exploiting historical relationships of clauses and variables in local search for satisfiability – (poster presentation), in: Proc. of SAT-12, 2012, pp. 479–480.
[29] Chu Min Li, Yu Li, Description of sattime2012, in: Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions, 2012, p. 53.
[30] Chuan Luo, Shaowei Cai, Wei Wu, Kaile Su, Focused random walk with configuration checking and break minimum for satisfiability, in: Proc. of CP-13, 2013, pp. 481–496.
[31] Chuan Luo, Kaile Su, Shaowei Cai, Improving local search for random 3-SAT using quantitative configuration checking, in: Proc. of ECAI-12, 2012, pp. 570–575.
[32] Bertrand Mazure, Lakhdar Sais, Éric Grégoire, Tabu search for SAT, in: Proc. of AAAI-97, 1997, pp. 281–285.
[33] Stephan Mertens, Marc Mézard, Riccardo Zecchina, Threshold values of random k-SAT from the cavity method, Random Struct. Algorithms 28 (3) (2006) 340–373.
[34] David A. McAllester, Bart Selman, Henry A. Kautz, Evidence for invariants in local search, in: Proc. of AAAI-97, 1997, pp. 321–326.
[35] Marc Mézard, Passing messages between disciplines, Science 301 (2003) 1685–1686.
[36] Wil Michiels, Emile H.L. Aarts, Jan H.M. Korst, Theoretical Aspects of Local Search, Springer, 2007.
[37] Christos H. Papadimitriou, On selecting a satisfying truth assignment, in: Proc. of FOCS-91, 1991, pp. 163–169.
[38] Ramamohan Paturi, Pavel Pudlák, Michael E. Saks, Francis Zane, An improved exponential-time algorithm for k-SAT, J. ACM 52 (3) (2005) 337–364.
[39] Duc Nghia Pham, Thach-Thao Duong, Abdul Sattar, Trap avoidance in local search using pseudo-conflict learning, in: Proc. of AAAI-12, 2012, pp. 542–548.
[40] Olivier Roussel, Description of ppfolio 2012, in: Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions, 2012, p. 46.
[41] Said Salhi, Defining tabu list size and aspiration criterion within tabu search methods, Computers Oper. Res. 29 (1) (2002) 67–86.
[42] Sakari Seitz, Mikko Alava, Pekka Orponen, Focused local search for random 3-satisfiability, J. Stat. Mech. (2005) P06006.
[43] Bart Selman, Henry A. Kautz, Bram Cohen, Noise strategies for improving local search, in: Proc. of AAAI-94, 1994, pp. 337–343.
[44] Kevin Smyth, Holger H. Hoos, Thomas Stützle, Iterated robust tabu search for MAX-SAT, in: Proc. of Canadian Conference on AI, 2003, pp. 129–144.
[45] Robert Stelzmann, The stochastic local search solver: ssa, in: Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions, 2012, p. 63.
[46] John Thornton, Duc Nghia Pham, Stuart Bain, Valnir Ferreira Jr., Additive versus multiplicative clause weighting for SAT, in: Proc. of AAAI-04, 2004, pp. 191–196.
[47] Dave A.D. Tompkins, Adrian Balint, Holger H. Hoos, Captain jack: New variable selection heuristics in local search for SAT, in: Proc. of SAT-11, 2011, pp. 302–316.
[48] Dave A.D. Tompkins, Holger H. Hoos, Dynamic scoring functions with variable expressions: New SLS methods for solving SAT, in: Proc. of SAT-10, 2010, pp. 278–292.
[49] Andreas Wotzlaw, Alexander van der Grinten, Ewald Speckenmeyer, Stefan Porschen, pfolioUZK: Solver description, in: Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions, 2012, p. 45.
[50] Lin Xu, Frank Hutter, Jonathan Shen, Holger H. Hoos, Kevin Leyton-Brown, Satzilla2012: Improved algorithm selection based on cost-sensitive classification models, in: Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions, 2012, p. 57.