SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2017, Vol. 60 092109:1–092109:11
doi: 10.1007/s11432-016-0258-4

A randomized diversification strategy for solving satisfiability problem with long clauses

Jian GAO[1,2,3], Ruizhi LI[1] & Minghao YIN[1*]

[1]*College of Computer Science, Northeast Normal University, Changchun* 130024*, China;*
[2]*College of Information Science and Technology, Dalian Maritime University, Dalian* 116026*, China;*
[3]*Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun* 130012*, China*

Received August 4, 2016; accepted September 12, 2016; published online May 11, 2017

Abstract Satisfiability problem (SAT) is a central problem in artificial intelligence due to its computational complexity and usefulness in industrial applications. Stochastic local search (SLS) algorithms are powerful to solve hard instances of satisfiability problems, among which CScoreSAT is proposed for solving SAT instances with long clauses by using greedy mode and diversification mode. In this paper, we present a randomized variable selection strategy to improve efficiency of the diversification mode, and thus propose a new SLS algorithm. We perform a number of experiments to evaluate the new algorithm comparing with the recently proposed algorithms, and show that our algorithm is comparative with others for solving random instances near the phase transition threshold.

Keywords SAT, local search, randomized diversification strategy, phase transition, long clause

Citation Gao J, Li R Z, Yin M H. A randomized diversification strategy for solving satisfiability problem with long clauses. Sci China Inf Sci, 2017, 60(9): 092109, doi: 10.1007/s11432-016-0258-4

1 Introduction

The Boolean satisfiability problem (SAT) is a basic NP-complete problem. It is one of the most frequently studied problems among combinatorial problems; it plays an important role in both computer science and artificial intelligence, since a number of academic and industrial problems can be transformed into SAT problems and then solved by a number of efficient SAT solvers. The task of solving SAT problem is to find an assignment to all Boolean variables of a given instance. In general cases, this task is extremely difficult for most instances due to the computational complexity reason.

To solve the SAT problems, there have been two main kinds of search algorithms: complete algorithms and local search algorithms. Usually, Davis-Putnam-Logemann-Loveland (DPLL) [1, 2] is a basic and efficient algorithm among the complete approaches, and in recent studies clause learning techniques [3, 4] have been employed to improve efficiency of complete search algorithms. Those algorithms can find a solution or all solutions if the SAT instance is satisfiable; otherwise, it can prove that the given instance is unsatisfiable. They may be comparatively slow when they are used to solve large-scaled instances and

* Corresponding author (email: ymh@nenu.edu.cn)

© Science China Press and Springer-Verlag Berlin Heidelberg 2017

random instances at phase transition point. On the contrary, another kind of algorithm, usually called local search algorithm [5, 6], may probably find a solution of satisfiable hard instances in a reasonable time though the drawback of them is that they are not able to prove unsatisfiability.

To evaluate search algorithms for SAT problems, industrial instances and randomly generated instances are usually employed as benchmarks. To solve those SAT instances from industrial areas, backtrack algorithms with sophisticated heuristics have been proposed to deal with special structures in the instances. However, random instances are generated with uniform distribution, which are often hard to solve by systemic backtrack algorithms because there is little structure information in the instances that can be explored. Moreover, some of those random instances have a few solutions so that it is difficult for the algorithms to find a solution [7].

In fact, the number of solutions in a randomly generated instance is determined by control parameters in a generation model, and control parameters are chosen based on phase transition theory [8–10], which specifies hardness of random instances. Random instances are also used to investigate theory analysis of computational complexity, which is another significant topic in SAT problems [7, 9, 10], attracting much attention as it is of great interest. The most referred phase transition in SAT problems is the solutionable phase transition, which separates soluble-to-insoluble areas on the phase transition point. It is also associated with an easy-hard-easy pattern [11], i.e., algorithms will spend the longest time on solving random instances at phase transition point, so those random instances generated at the transition point are hardest SAT instances to solve.

Since the local search algorithm GSAT proposed by Selman et al. [12], a number of local search algorithms with stochastic approaches, usually called stochastic local search (SLS) algorithms, have been presented [13–20]. They have revealed good performance for solving random SAT instances, where WalkSAT [6] is one of the most investigated algorithms. In recent years, there is great interest in improving SLS algorithms to solve hard random instances, and we have viewed many contributions on new strategies of SLS algorithms. Many heuristic strategies have been discussed. Among them clause weighting technique [13, 14] is an old but efficient method. It is employed in many SLS algorithms. As an example, Hutter et al. [13] proposed the scaling and probabilistic smoothing (SAPS) algorithm. With this method, heuristic approaches were developed to make decision on variable selection for flipping, where the variable having minimum value of change in sum of false clause weights caused by the flipping is usually selected. Testing on solving SAT instances indicates that those techniques can make SLS algorithms efficiently. Later, Thornton et al. [14] developed a pure additive weighting scheme (PAWS) algorithm, reporting that PAWS performs better on solving hard instances.

Recently, CScoreSAT has been proposed by Cai et al. [21, 22], which is also an SLS algorithm with clause weighting technique. To deal with SAT instances with long clauses, two modes are employed in the algorithm: greedy mode and diversification mode. The greedy mode tries to select the right variables to flip to decrease the number of false clauses, whereas the diversification mode tries to escape the local area when SLS reaches a local optimum. Besides, Cai et al. [21] also proposed the HScoreSAT algorithm, which is a modified version from CScoreSAT. Both algorithms are particularly designed to solve SAT problems with long clauses. They have shown that HScoreSAT performs better than CScoreSAT for random instances at the phase transition point, whereas CScoreSAT is faster than HScoreSAT for instances below the phase transition point. The solver probSAT, another SLS-based algorithm, was also extended in order to solve SAT instances with long clauses [23]. It utilizes a probability selection to determine the variable for flipping, where exponential functions are used to choose the next variable to flip. This method highly depends on control parameters they set. Very recently, Liu and Papakonstantinou [24] have employed polynomial valuation for variable selection, and proposed an algorithm called polyLS. They have shown that polyLS outperforms the state-of-the-art algorithms by testing benchmark instances generated exactly at the phase transition points.

As we have mentioned, the instances generated exactly at or near the phase transition points are always very hard to solve, and improving searching efficiency is difficult for those instances, so developing new strategies to improve SLS algorithms is still a challenge issue in the SAT community. Most researches focus on making improvement on solving instances generated exactly at the phase transition points,

but not all those algorithms perform well on solving hard instances near the phase transition point. In this paper, we concentrate on developing a novel diversification heuristic to improve the CScoreSAT algorithm with the aim at solving k-SAT instances near the phase transition point. A novel heuristic strategy in the diversification mode is presented using a randomized approach, which is incorporated into the CScoreSAT algorithm instead of the original diversification method. The key feature of the novel method is that it only considers a part of variables in a clause. State in another way, it randomly selects several variables and compares scores in order to find the best one among them for flipping rather than selecting the best variable with smallest score in a clause for flipping. This new diversification strategy is more robust compared with the method selecting the best variable, and has ability to lead the search algorithm to different directions when trapping in the same local optimum. We analyze the results of experiments on randomly generated instances near the phase transition point intensively. It can be seen that the algorithm with the novel randomized diversification strategy shows better performance than the algorithm with original strategy, where results of runtimes indicate that the proposed algorithm is more efficient than CScoreSAT, and the number of instances solved within the limited time by the proposed algorithm is also more than that of CScoreSAT. In addition, we also compare our algorithm with other existing SAT algorithms showing the advantages of our algorithm.

2 Preliminaries

In this section, we first introduce the propositional satisfiability problem and its random generated model. Then, SLS algorithms as well as some main related techniques are reviewed.

2.1 SAT model

We consider propositional formula in conjunctive normal form (CNF) in this paper. Here we give some definitions about CNF. A literal is a Boolean variable or its negation, and a clause is a disjunction of literals, where the number of literals in the clause is called the length of the clause. A propositional formula F in CNF is with the form of a conjunction of clauses. The notation k-SAT often refers to the SAT problems whose clause length is k. It is well known that 2-SAT can be solved in polynomial time, while k-SAT ($k > 2$) is NP-complete.

Usually, in order to generate hard SAT instances for testing various SAT solvers, randomly generated instances are considered. Those instances are produced randomly with some generation models. A frequently used model is described as follows [12, 25]: three parameters k, n, m are employed to generate SAT instances, where k is the length of clauses, n is the number of variables, and m is the number of clauses. A ratio r is defined as $r = m/n$. To obtain a random SAT instance, we should generate m clauses, where each clause is generated by selecting k variables without replacement from n variables and negating each variable with probability 0.5, and thus clause generation is with uniform probability distribution, and all clauses generated are of the same length k. Compared with SAT instances from industrial areas, using this model, we can obtain a number of SAT instances, and we can also adjust the control parameters r and n to generate instances whose size and difficulty are desirable. Theoretical results on the random k-SAT model have shown that phase transition occurs at $r = 4.267$ with $k = 3$, and thus instances generated with this parameter are hardest to solve. Moreover, hardest instances are also with $r = 9.931$ for 4-SAT, with $r = 21.117$ for 5-SAT, with $r = 43.37$ for 6-SAT, and with $r = 87.79$ for 7-SAT. Therefore, random instances are good benchmarks to evaluate various SAT algorithms, especially for SLS algorithms. To evaluate effectiveness of SLS algorithms and to show ability of solving hard instances, most recent studies have performed testing on random instances.

2.2 SLS algorithms

An SLS algorithm often starts from a random assignment as initialization, and then it carries out iterations of flipping variables until a solution is found or the prefixed limited time is achieved. It has been pointed out by many authors that heuristic methods used for selecting a variable for flipping in each step impacts

on efficiency of the entire SLS algorithm. To design efficient heuristic strategies, the functions make(x) and break(x) are usually used to decide which variable will be selected to be flipped in the next iteration. For instance, walkSAT and probSAT employ break(x) to make decision on which variable will be selected. Moreover, both of make(x) and break(x) are incorporated into the SAT solver CScoreSAT. Besides, tie breaking strategies are also used in the heuristics when some variable scores are of same value. Moreover, timestamp is a useful strategy to break ties in many cases.

Clause weighting technique is also an important technique. By making penalty on unsatisfied clauses and increasing weights of those clauses, SLS algorithms can avoid trapping in some local optima. Clause weighting algorithms initialize weights of all clauses to one. Score functions are calculated based on those weights. In a step when it reaches a local optimal point, clause weighting technique usually increases weights of unsatisfied clauses to enhance the importance of those clauses. So variables in clauses difficult to be satisfied will be flipped with high probability. This technique has been considered in many SAT solvers.

In addition, there are some other techniques used in the SLS SAT solvers. An important one among those techniques is the configuration checking (CC) strategy [26], first proposed by Cai et al. [27] for vertex cover problem, and later it was incorporated into SLS algorithms for k-SAT problems and set covering problems, and has been employed in a series of SLS algorithms, such as CScoreSAT, CCASat [19], DCCASat [28], and uSLC [29]. CC strategy forbids variables to be flipped if their neighborhoods are not changed, and thus avoid frequent flips on a certain variable that may cause a search cycle. Before their work, previous SLS algorithms had not employed this strategy while they flip variables only according to the properties of variables rather than flipping information of their neighbors.

2.3 CScoreSAT

CScoreSAT algorithm is a SLS algorithm with clause weighting techniques and CC strategy, which aims at particularly solving SAT instances with clause length greater than 3. Like most SLS algorithms, CScoreSAT is designed as a two-mode algorithm. However, different from previous ones, to deal with long clauses, the second level properties are considered in CScoreSATalgorithm. More specifically, function make2(x) is defined as the number of 1-satisfied clauses that will change to 2-satisfied after flipping x; and function break2(x) is the number of 2-satisfied clauses that will change to 1-satisfied, where the 1-satisfied clause denotes there is only one true literal in a clause while a clause is a 2-satisfied clause if there are 2 true literals. Using function make2(x) and break2(x), variable selection can make decision with more information of clause states compared to using functions make(x) and break(x). This method plays a crucial role in the CScoreSAT solver.

Timestamp is another measurement to determine the variable to be flipped, especially for breaking ties. We use age(x) to count steps from last flip of the variable x to the current step. Heuristic strategy of CScoreSAT will select the variable with largest age(x) if two variables gain the same score value.

Based on the make and break functions, cscore function is defined as the following formula:

$$\text{cscore}(x) = \text{score}(x) + \text{score2}(x)/d, \tag{1}$$

where d is a control parameter taking positive integer, $\text{score}(x) = \text{make}(x) - \text{break}(x)$, and $\text{score2}(x) = \text{make2}(x) - \text{break2}(x)$. And hscore function is defined as

$$\text{hsocre}(x) = \text{cscore}(x) + \text{age}(x)/\beta = \text{score}(x) + \text{score2}(x)/d + \text{age}(x)/\beta, \tag{2}$$

where control parameters d and β are positive integers.

In the greedy mode of CScoreSAT, hscore function is the heuristic strategy for selecting the best variable, where the variable with minimum hscore is flipped in the step if the total score can be improved after flipping. In the case that searching reaches a local optimum, the algorithm alters to the diversification mode, where cscore function is used as a criterion for determining the flipped variable. It is noted that the best one is selected with breaking tie by timestamp.

The method for clause weight updating like the method in PAWS, proposed in [14], is used in CScoreSAT. It will increase weights of false clauses by 1, and weight smoothing will be performed with a certain probability and with the case that the number of clauses with weight more than 1 achieves the threshold.

With the techniques above, CScoreSAT is described as follows Algorithm 1.

Algorithm 1 CScoreSAT

```
1: Input CNF-formula F, maxSteps;
2: output a solution or "unknown";
3: α ← randomly generated truth assignment;
4: for step ← 1 to maxSteps do
5:    if α satisfies F then
6:       return α;
7:    end if
8:    if S = {x|x is comprehensively decreasing and configuration changed} is not empty then
9:       v ← a variable in S with the greatest cscore, breaking ties in favor of the one with largest age;
10:   else
11:      update clause weights according to PAWS;
12:      pick a random falsified clause C;
13:      v ← the variable in C with the greatest hscore;
14:   end if
      α ← α with v flipped;
15: end for
16: return "unknown".
```

Like most SAT solvers, CScoreSAT randomly generates an initial solution, and iterates within a prefixed number of steps. A step is composed of two main operations: picking up a variable and flipping the variable in the implementation of CScoreSAT solver. The operation of picking up a variable (Lines 6 to 11) finds the best variable if flipping it can make improvement on total score, which is task of the greedy mode; otherwise it returns best variable in the random chosen unsatisfied clause, which is task of the diversification mode. The operation of flipping a variable changes the value and then updates score values related to the variables. Besides, clause weights will be updated or be smoothed if the diversification mode is performed. The algorithm outputs a solution to the given instance or reports no solution has been found.

Most SLS solvers always take random SAT instances exactly at the phase transition point as benchmark. Some of them can achieve good performance to solve those instances, because strategies and control parameters are optimized according to those benchmarks, whereas strategies they used may not be effective enough to obtain satisfactory results when solving other instances, even solving random instances near phase transitions point. As an example, HScoreSAT solver is an enhanced solver for hardest instances based on hscore function. Though HScoreSAT solver can perform better than CScoreSAT for solving phase transition instances, it is shown that CScoreSAT can solve instances below phase transition point more efficiently than HScoreSAT. Cai et al. [22] also conjectured there exists a boundary ratio for each k such that below boundary ratio CScoreSAT outperforms HScoreSAT.

3 The new strategy and algorithm

In this section, we present the randomized strategy in details, and discuss implementation of the strategy. After that, we describe the entire algorithm based on this new strategy.

3.1 The randomized diversification strategy

Though it has been shown that some new SLS algorithms can achieve better performance than CScoreSAT on solving random instances at the phase transition point, CScoreSAT is still competitive for satisfiable random instances near the point. The key feature to make CScoreSAT so efficient is the heuristic functions hscore and cscore. With those two functions, right variables are probably selected to make good moves in both greedy and diversification modes, so it seems hard to further improve the searching efficiency by

modifying those two heuristic functions. However, to make CScoreSAT more efficient for the instances near the phase transition point, in this section, we propose a new variable selection approach in the diversification mode.

As we have mentioned in the previous section, CScoreSAT randomly selects an unsatisfiable clause and then finds the variable with the highest cscore in the clause. This method can make a good move when it reaches a local optimal point, and then searches its neighborhoods. However, if a local optimal point is revisited many times, this method may pick up the same variable if the same clause is chosen if there is only one or several unsatisfiable clauses in local optima. It is also clear that the fewer of unsatisfiable clauses, the higher probability of the same variable will be chosen. Though it finds the best variable such that flipping that variable makes the smallest increment of the total score, it may always choose same variables, so cycling issue cannot be settled in the diversification mode. To some extent, this issue will occur more frequently as the algorithm finds better assignments with fewer unsatisfiable clauses. Therefore, CScoreSAT may suffer this issue and may flip some variables too frequently, and thus make more cycling moves. Based on this consideration, we present a randomized strategy. The technique described in the followings aims at improving diversification strategy so as to avoid trapping in a local area and thus exploring more solution space.

As we discussed above, the existing method used in CScoreSAT algorithm only returns the best variable. We make it more robust. Specifically, after choosing an unsatisfiable clause, the robust method selects R variables and finds the best one among them, where hscore function is also the measure of those variables. Duplicated variables may be selected. The times of selecting variables in a clause is generated randomly for each step. To choose more variables with better score for flipping, we set the random number ranging from $\lfloor k/2 \rfloor + 1$ to k. With the discussion above, we proposed the new randomized strategy. The following procedure shows details of the variable selection strategy in the diversification mode:

Step 1: choose an unsatisfiable clause C and generate the random integer R ranging from $\lfloor k/2 \rfloor + 1$ to k;

Step 2: select a variable from the clause C and compute its hscore;

Step 3: record the variable with the greatest hscore as v_b among all selected variables;

Step 4: repeat Steps 2 and 3 R times, and return the variable v_b.

For the implementation aspect, we also use the function hscore to evaluate variables in the new strategy. It is also noted that the modified strategy is easy to implement as we can modify the diversification part: after determining the unsatisfiable clause, it uses the new strategy, and it generates the number of variable selecting times, denoted by R; and then it repeats selecting of variables R times; at last the best one among them is returned. It is also seen that the modification does not take much external time to compute the next variable, as the number of variables picked up is smaller than k, so computation of function hscore do not increase much time compared with original strategy.

3.2 The entire algorithm

In the following, we describe the complete algorithm proposed for solving SAT problems with long clauses. The new algorithm employs the techniques in the original CScoreSAT algorithm, such as the greedy mode, clause weighting, and CC strategy, while it also combines the randomized variable strategy in the diversification mode. Hence, we can modify the CScoreSAT algorithm to propose the new one, called RCScoreSAT. Recall the CScoreSAT algorithm, the variable with the greatest hscore will be selected (Line 11). In RCScoreSAT, we use the randomized variable strategy described above instead of original strategy, so Line 11 will be modified to the following statement:

$v \leftarrow$the variable selected by the randomized strategy.

RCScoreSAT also calculates function hscore and function cscore to evaluate variables, which is same as CScoreSAT. It is noted that the new strategy is easy to implement and the algorithm only introduces one extra control parameter that determines the number of variables considered.

4 Experiment evaluations

In this section, we perform an intensive computational experiment to evaluate the new variable selection strategy and the SAT algorithm based on the strategy. To study the effectiveness of the new strategy we propose and provide a fair comparison, we incorporate the strategy into the CScoreSAT solver, which is developed by Cai et al. with C++ language. We set parameters derived from the original CScoreSAT algorithm same as values reported in the paper [22]. We perform all experiments parallel on a workstation with two Intel Xeon E5-2637v3 (3.5 GHz) CPUs, and 32 GB RAM, running Ubuntu Linux 15.04. Computational experiments in the followings are divided into 3 parts: the first part analyzes the control parameter of the new strategy; the second part analyzes average CPU run times cost on a single step; and the last part compares computational results of the new algorithm with the results produced by CScoreSAT and other existing algorithms.

To show the performance of the new algorithm on solving instances near phase transition areas, we generate SAT instances using the random generator, which was also used in SAT Competition 2013 (available at http://sourceforge.net/projects/ksatgenerator/). We can obtain SAT instances without duplicate clauses by the generator. Considering 3-SAT instances are usually solved by other efficient SLS algorithms rather than CScoreSAT, we only generate instances with long clauses in this paper, i.e., $k = 4, 5, 6, 7$, respectively. For each k, we choose the ratio r near the phase transition area as does in [22], where the first 9 values are selected that are near the phase transition point, whereas the value at the point is not chosen since the improvement of this paper aims at solving instances near the point. Though the value of the ratio r impacts on CPU runtime greatly, CPU runtime of an instance also depends on the number of variables (n). We take $n = 3000$, $n = 550$, $n = 350$, and $n = 150$ to generate instances for $k = 4, 5, 6, 7$, respectively. Ten instances are generated for each r and each k, and thus there are 360 instances totally, which will be taken as benchmark to test our proposed algorithm and other algorithms.

Moreover, huge groups of benchmarks in SAT race 2013 and 2014 are also taken as test instances. For each k, there are 6 and 15 large-size instances in benchmark of 2013 and 2014 respectively.

4.1 Analyze on the new randomized strategy

We first analyze the number of variables selected for each clause. To show good performance of the method ranging the number R from $\lfloor k/2 \rfloor + 1$ to k, we take two other parameters, the first one ranging the random number R from 1 to $\lfloor k/2 \rfloor$denoted by RCScorceSAT1, and the second one ranging the number from k to $\lfloor 3k/2 \rfloor$ denoted by RCScorceSAT2. Hence, three groups of results are obtained after solving the total 360 random instances with the cutoff time 1000 s. Table 1 shows the computational results.

The par10 run time is computed and shown in the table. The par10 run time is always used for evaluating other SAT algorithms, where unsuccessful runs within limited time will be penalized by 10 times of the limited CPU time. From this table, we can see that the RCScorceSAT preforms better than the other two contrast algorithms on both the number of total successful runs and average run times for all $k = 4, 5, 6, 7$. For most values of r, RCScorceSAT also performs better. This indicates bad variables may be selected if we choose variable with 0 to $\lfloor k/2 \rfloor$ times, whereas the best variables may always be selected if the number is bigger than k, so we set the range of the random number ranging from $\lfloor k/2 \rfloor + 1$ to k.

4.2 Comparison with existing algorithms

In this subsection, we evaluate efficiency of the new algorithm. Comparisons are made between the new one and CScoreSAT [21], probSAT [23] and CSCCSat [30]. Same as the previous section, we also test those algorithms on SAT instances that are generated in the former subsection, limiting to 1000 s for each run. Codes of probSAT and CSCCSat are also downloaded from the website of SAT competition 2014. We compile them settings all parameters default same as these used in the competition. We perform experiments same as the experiments in former subsection, where each instance is solved 5 runs

Table 1 Comparison of randomized strategies

		RCScoreSAT		RCScoreSAT1		RCScoreSAT2	
		Avg.	#Succ.	Avg.	#Succ.	Avg.	#Succ.
$k=4$	$r=9.000$	0.2428	50	0.3104	50	0.2724	50
	$r=9.121$	0.3762	50	0.4706	50	0.461	50
	$r=9.223$	0.7174	50	0.8094	50	0.8446	50
	$r=9.324$	1.2438	50	1.678	50	1.3054	50
	$r=9.425$	2.751	50	2.801	50	3.4376	50
	$r=9.526$	8.277	50	10.9864	50	11.4096	50
	$r=9.627$	30.3548	50	41.5196	50	41.653	50
	$r=9.729$	771.9118	47	982.9466	46	1576.488	43
	$r=9.830$	8459.8	8	8669.37	7	8857.994	6
	Avg./total	1030.631	405	1078.988	403	1165.985	399
$k=5$	$r=20.000$	2.2022	50	2.3616	50	3.3704	50
	$r=20.155$	5.423	50	7.526	50	5.9364	50
	$r=20.275$	17.1128	50	21.997	50	23.5836	50
	$r=20.395$	39.2522	50	89.2342	50	41.5158	50
	$r=20.516$	1042.579	45	872.529	46	1075.462	45
	$r=20.636$	1336.254	44	2120.392	40	1891.133	41
	$r=20.756$	3535.204	33	4342.104	29	3929.005	31
	$r=20.876$	6518.784	18	6096.655	20	6747.939	17
	$r=20.997$	9033.17	5	9616.664	2	9245.974	4
	Avg./total	2392.22	345	2574.385	337	2551.547	338
$k=6$	$r=40.000$	1.8736	50	2.5686	50	2.1868	50
	$r=40.674$	5.4836	50	8.701	50	7.5538	50
	$r=41.011$	18.3818	50	25.6618	50	14.6598	50
	$r=41.348$	28.558	50	44.1718	50	43.7034	50
	$r=41.685$	382.0158	49	341.2936	49	730.8804	47
	$r=42.022$	3117.932	35	3006.646	36	3106.705	35
	$r=42.359$	6121.675	20	6557.346	18	5983.66	21
	$r=42.696$	8267.004	9	8493.843	8	8333.469	9
	$r=43.033$	9224.432	4	9437.054	3	9621.87	2
	Avg./total	3018.595	317	3101.921	314	3093.854	314
$k=7$	$r=85.000$	915.5352	46	936.1416	46	1332.871	44
	$r=85.558$	982.968	46	1030.254	46	633.829	48
	$r=85.837$	1941.392	41	1609.236	43	1948.94	41
	$r=86.116$	1747.858	42	2383.182	39	2351.993	39
	$r=86.395$	2134.722	40	2972.488	36	2214.348	40
	$r=86.674$	3312.208	34	3015.547	36	3139.122	35
	$r=86.953$	4325.604	29	4302.681	29	4688.863	27
	$r=87.232$	4494.174	28	4873.474	26	4931.854	26
	$r=87.511$	5863.802	21	5897.31	21	5914.119	21
	Avg./total	2857.585	327	3002.257	322	3017.326	321

by each solver. The number of successful runs and average runtimes of those algorithms are calculated, and comparative results are listed in Table 2.

From Table 2, we can see that RCScorceSAT achieves best average performance of all groups among those SAT solvers. It improves the performance of CScoreSAT greatly for each k. For each r, RCScorceSAT can achieve equal to or more successful runs than CScoreSAT. RCScorceSAT also performs better on run times when RCScorceSAT and CScoreSAT have the same number of successful runs. Moreover, probSAT performs best on 4-SAT instances, but it is not good to solve others as k increases. RCScorceSAT performs best on 5-SAT, 6-SAT and 7-SAT. probSAT performs better than CScoreSAT on 5-SAT but worse than CScoreSAT on 6-SAT and 7-SAT. This phenomenon was also reported in [23] for solving

Table 2 Comparative results with existing algorithms

		probSAT		CSCCSat		CScoreSAT		RCScoreSAT	
		Avg.	#Succ.	Avg.	#Succ.	Avg.	#Succ.	Avg.	#Succ.
$k=4$	$r=9.000$	0.09	50	0.09	50	0.42	50	0.24	50
	$r=9.121$	0.17	50	0.14	50	0.80	50	0.38	50
	$r=9.223$	0.20	50	0.28	50	1.21	50	0.72	50
	$r=9.324$	0.37	50	0.59	50	2.52	50	1.24	50
	$r=9.425$	1.75	50	4.04	50	4.87	50	2.75	50
	$r=9.526$	3.65	50	11.62	50	20.54	50	8.28	50
	$r=9.627$	21.20	50	48.96	50	78.24	50	30.35	50
	$r=9.729$	924.33	46	1245.78	45	1615.92	42	771.91	47
	$r=9.830$	7501.65	13	9439.60	3	9647.33	1	8459.80	8
	Avg./total	939.27	409	1194.57	398	1263.54	393	1030.63	405
$k=5$	$r=20.000$	4.08	50	9.31	50	4.89	50	2.20	50
	$r=20.155$	13.40	50	12.29	50	13.51	50	5.42	50
	$r=20.275$	30.00	50	35.20	50	19.45	50	17.11	50
	$r=20.395$	460.59	48	64.94	50	277.02	49	39.25	50
	$r=20.516$	920.08	46	902.43	46	1094.66	45	1042.58	45
	$r=20.636$	2355.57	39	2153.10	40	3141.62	35	1336.25	44
	$r=20.756$	5293.49	24	4570.53	28	5491.06	23	3535.20	33
	$r=20.876$	7118.93	15	6135.60	20	7702.45	12	6518.78	18
	$r=20.997$	10000.00	0	9608.52	2	9806.07	1	9033.17	5
	Avg./total	2910.68	322	2610.21	336	3061.19	315	2392.22	345
$k=6$	$r=40.000$	21.51	50	21.21	50	2.93	50	1.87	50
	$r=40.674$	144.76	50	288.01	49	13.06	50	5.48	50
	$r=41.011$	1400.17	44	1357.09	44	45.22	50	18.38	50
	$r=41.348$	4146.96	30	606.85	48	279.15	49	28.56	50
	$r=41.685$	6568.72	18	3256.38	35	1181.82	45	382.02	49
	$r=42.022$	8660.78	7	6130.17	20	3776.20	32	3117.93	35
	$r=42.359$	9431.45	3	8098.15	10	8263.23	9	6121.67	20
	$r=42.696$	10000.00	0	9421.03	3	9601.39	2	8267.00	9
	$r=43.033$	10000.00	0	9629.11	2	9057.76	5	9224.43	4
	Avg./total	5597.15	202	4312.00	261	3580.08	292	3018.60	317
$k=7$	$r=85.000$	4784.90	27	1128.84	45	1763.02	42	915.54	46
	$r=85.558$	4816.80	27	998.96	46	2921.53	36	982.97	46
	$r=85.837$	5175.96	25	2011.31	41	2336.43	39	1941.39	41
	$r=86.116$	6187.39	20	2387.12	39	2439.62	39	1747.86	42
	$r=86.395$	6770.92	17	2031.96	41	2757.06	37	2134.72	40
	$r=86.674$	5835.77	22	3371.87	34	3736.29	32	3312.21	34
	$r=86.953$	7299.66	14	4290.29	29	4910.41	26	4325.60	29
	$r=87.232$	6512.71	18	4400.07	29	4766.73	27	4494.17	28
	$r=87.511$	8272.29	9	6486.55	18	6040.26	20	5863.80	21
	Avg./total	6184.05	179	3011.89	322	3519.04	298	2857.58	327

benchmark instances from SAT competitions, where probSAT performs better than CScoreSAT on 5-SAT but CScoreSAT performs better on 7-SAT. CSCCSat performs slightly better than CScoreSAT on total average runtime but it is worse than RCScorceSAT. RCScorceSAT is better than CSCCSat for most values of r. Particularly, RCScorceSAT has good ability to solve instances when r decreases compared to CSCCSat except 4-SAT.

Furthermore, we take huge-size instances from SAT Competition 2013 (SC-13) and 2014 (SC-14) to test our algorithm and the original CScoreSAT. Like pervious experiment, we run 5 times for each instance, but the cutoff time is set to 2000 s. Table 3 indicates average run times and the number of successful runs grouped by k. As can be seen from it, RCScoreSAT performs better than CScoreSAT since it achieves

Table 3 Comparative results on huge-size instances

	CScoreSAT		RCScoreSAT	
	Avg.	#Succ.	Avg.	#Succ.
SC-13 $k = 4$	7111.00	20	4473.53	24
SC-13 $k = 5$	10095.09	15	6968.89	20
SC-13 $k = 6$	10178.51	15	10085.21	15
SC-13 $k = 7$	16675.09	5	13555.42	10
SC-14 $k = 4$	14192.87	23	13642.79	25
SC-14 $k = 5$	13556.33	25	12769.36	28
SC-14 $k = 6$	10797.84	35	10266.73	37
SC-14 $k = 7$	13719.65	24	12412.49	29
Avg./total	12040.80	162	10521.80	188

more successful runs and requires less CPU runtime for each group of instances.

5 Conclusion

The SAT problem is one of central issues in artificial intelligence, and is the prototype problem in NP-complete. Many backtracking algorithms and SLS algorithms for SAT have been presented. Recently, SLS algorithms, such as CScoreSAT solver and probSAT solver, have revealed good performance to solve hard SAT instances. CScoreSAT performs well on instances with long clauses, where greedy mode and diversification mode are important components in the SAT solvers. In this paper, we proposed a new diversification strategy to improve the performance of CScoreSAT. Our contributions are summarized as follows: In diversification mode, we modify the original strategy by using a more robust way for selecting variables rather than choosing the best one to flip. This method can make different flips when the search algorithm reaches the same local optimum so that diversification is enhanced compared to original strategy. Then, the new diversification strategy is incorporated in CScoreSAT solver. A number of computational experiments are performed with analyzing on control parameters and CPU run time. Also, comparisons are made with other existing SLS algorithms. We have shown that the new proposed SLS algorithm improves performance on solving instances near the phase transition point.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 61370156, 61402070, 61503074), Program for New Century Excellent Talents in University (Grant No. NCET-13-0724), and Natural Science Foundation of Liaoning Province (Grant No. 2015020023).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Davis M, Putnam H. A computing procedure for quantification theory. J ACM, 1960, 7: 201–215
2 Davis M, Logemann G, Loveland D. A machine program for theorem proving. Commun ACM, 1962, 5: 394–397
3 Marques-Silva J P, Sakallah K A. GRASP: a search algorithm for propositional satisfiability. IEEE Trans Comput, 1999, 48: 506–521
4 Zhang L, Madigan C F, Moskewicz M H, et al. Efficient conflict driven learning in a Booleansatisfiability solver. In: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design. Piscataway: IEEE Press, 2001. 279–285
5 Selman B, Kautz H A, Cohen B. Local search strategies for satisfiability testing. Discrete Math Theor, 1996, 26: 521–532
6 Selma B, Kautz H A, Cohen B. Noise strategies for improving local search. In: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94). San Francisco: AAAI Press, 1994. 337–343
7 Achlioptas D, Naor A, Peres Y. Rigorous location of phase transitions in hard optimization problems. Nature, 2005, 435: 759–764
8 Cheeseman P, Kanefsky B, Taylor W M. Where the really hard problems are. In: Proceedings of the International Joint Conference on Artificial Intelligence. San Francisco: Morgan Kaufmann Publishers, 1991. 331–340
9 Xu K, Li W. Exact phase transitions in random constraint satisfaction problems. J Artif Intell Res, 2000, 12: 93–103

10 Xu K, Boussemart F, Hemery F, et al. Random constraint satisfaction: easy generation of hard (satisfiable) instances. Artif Intell, 2007, 171: 514–534

11 Mitchell D G, Selman B, Levesque H J. Hard and easy distributions of SAT problems. In: Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92). San Jose: AAAI Press, 1992. 459–465

12 Selman B, Levesque H, Mitchell D. A new method for solving hard satisfiability problems. In: Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92). San Jose: AAAI Press, 1992. 440–446

13 Hutter F, Tompkins D A D, Hoos H H. Scaling and probabilistic smoothing: efficient dynamic local search for SAT. In: Proceedings of the 8th International Conference on the Principles and Practice of Constraint Programming. Berlin: Springer, 2002. 233–248

14 Thornton J, Pham D N, Bain S, et al. Additive versus multiplicative clause weighting for SAT. In: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04). San Jose: AAAI Press, 2004. 191–196

15 Balint A, Schöning U. Choosing probability distributions for stochastic local search and the role of make versus break. In: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing. Berlin: Springer, 2012. 16–29

16 Li C M, Li Y. Satisfying versus falsifying in local search for satisfiability. In: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing. Berlin: Springer, 2012. 477–478

17 Wei W, Li C M, Zhang H. A switching criterion for intensification and diversification in local search for SAT. J Satisfiab Bool Model Comput, 2008, 4: 219–237

18 Duong T T, Pham D N, Sattar A, et al. Weight-enhanced diversification in stochastic local search for satisfiability. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, 2013

19 Cai S, Su K. Local search for Boolean Satisfiability with configuration checking and subscore. Artif Intell, 2013, 204: 75–98

20 Gableske O, Heule M. EagleUP: solving random 3-SAT using SLS with unit propagation. In: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing. Berlin: Springer, 2011. 367–368

21 Cai S, Su K. Comprehensive Score: towards efficient local search for SAT with long clauses. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, 2013

22 Cai S, Luo C, Su K. Scoring functions based on second level score for k-SAT with long clauses. J Artif Intell Res, 2014, 51: 413–441

23 Balint A, Biere A, Fröhlich A, et al. Improving implementation of SLS solvers for SAT and new heuristics for k-SAT with long clauses. In: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing. Berlin: Springer, 2014. 302–316

24 Liu S, Papakonstantinou P A. Local search for hard SAT formulas: The strength of the polynomial law. In: Proceedings of the 26th AAAI Conference on Artificial Intelligence, Phoenix, 2016

25 Liu J, Xu K. A novel weighting scheme for random k-SAT. Sci China Inf Sci, 2016, 59: 092101

26 Cai S, Su K. Configuration checking with aspiration in local search for SAT. In: Proceedings of the 26th AAAI Conference on Artificial Intelligence. Phoenix: AAAI Press, 2016

27 Cai S, Su K, Sattar A. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. Artif Intell, 2011, 175: 1672–1696

28 Luo C, Cai S, Wu W, et al. Double configuration checking in stochastic local search for satisfiability. In: Proceedings of the 28th AAAI Conference on Artificial Intelligence. Quebec City: AAAI Press, 2014. 2703–2709

29 Wang Y Y, Ouyang D T, Zhang L M, et al. A novel local search for unicost set covering problem using hyperedge configuration checking and weight diversity. Sci China Inf Sci, 2017, 60: 062103

30 Luo C, Cai S, Su K, et al. Clause states based configuration checking in local search for satisfiability. IEEE Trans Cybernetics, 2015, 45: 1014–1027