Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence

Configuration Checking with Aspiration in Local Search for SAT

Shaowei Cai[1,2] and **Kaile Su**[2,3*]
[1]Key laboratory of High Confidence Software Technologies, Peking University, Beijing, China
[2]Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Australia
[3]State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
shaowei_cai@126.com; sukl@pku.edu.cn

Abstract

An interesting strategy called configuration checking (CC) was recently proposed to handle the cycling problem in local search for Minimum Vertex Cover. A natural question is whether this CC strategy also works for SAT. The direct application of CC did not result in stochastic local search (SLS) algorithms that can compete with the current best SLS algorithms for SAT. In this paper, we propose a new heuristic based on CC for SLS algorithms for SAT, which is called configuration checking with aspiration (CCA). It is used to develop a new SLS algorithm called Swcca. The experiments on random 3-SAT instances show that Swcca significantly outperforms Sparrow2011, the winner of the random satisfiable category of the SAT Competition 2011, which is considered to be the best local search solver for random 3-SAT instances. Moreover, the experiments on structured instances show that Swcca is competitive with Sattime, the best local search solver for the crafted benchmark in the SAT Competition 2011.

Introduction

The propositional 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 (Kautz, Sabharwal, and Selman 2009). Two popular approaches for solving SAT are conflict driven clause learning (CDCL) and stochastic local search (SLS), and in this work we focus on the latter. While systematic approaches such as CDCL that perform considerable amounts of reasoning are often more useful on application instances, SLS is well known as the most effective approach for solving random satisfiable instances.

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 a variable is chosen and flipped. We use *pickVar* to denote the function for choosing the variable to be flipped. According to the heuristic used in *pickVar*, SLS algorithms can be divided into three categories: GSAT, WalkSAT and dynamic local search (DLS). The current best solvers, such as the three winners in SAT 2011 competition namely Sparrow2011 (Balint and Fröhlich 2010), Sattime2011 (Li and Li 2011), and EagleUP (Gableske and Heule 2011), are combinations of heuristics from these categories.

As stated in (Tompkins, Balint, and Hoos 2011), SLS algorithms for SAT usually work in two different modes, i.e., the greedy (intensification) mode and the diversification mode. In the greedy mode, they prefer variables whose flips can decrease the number of unsatisfied clauses; in the diversification mode, they tend to better explore the search space and avoid local optima, usually using randomized strategies and exploiting *diversification properties* of variables (Tompkins, Balint, and Hoos 2011) such as *age* and *flip count* to pick a variable for this aim. SLS algorithms for SAT usually utilize GSAT-like heuristics in the greedy mode and WalkSAT-like ones in the diversification mode.

An important issue for local search is the cycling problem, i.e., revisiting a candidate solution that has been visited recently (Michiels, Aarts, and Korst 2007). A strategy called configuration checking (CC) was recently proposed to deal with this issue, and was used to improve a state-of-the-art Minimum Vertex Cover (MVC) local search algorithm called EWLS (Cai, Su, and Chen 2010), which leads to the much more efficient SLS solver EWCC for MVC (Cai, Su, and Sattar 2011). A natural question is whether this CC strategy also works for SAT.

According to the CC strategy for MVC in (Cai, Su, and Sattar 2011), it is easy to develop 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. Actually, this has been used in an SLS algorithm called Swcc. However, Swcc cannot compete with the current best SLS solvers such as Sparrow2011 (Cai and Su 2011). In our opinion, the CC strategy is too strict for SLS algorithms for SAT, as it forbids all variables whose circumstance (i.e., the truth values of all its neighboring variables) has not changed since its last flip to be flipped, regardless of the benefit its flip can bring.

This work proposes a new heuristic based on CC for SLS algorithms for SAT. We name it Configuration Checking with Aspiration (CCA), as this new heuristic utilizes a mechanism which is inspired by the aspiration mechanism in tabu search (Glover 1990; Salhi 2002). According to CCA, there are two levels with different priorities in the greedy mode.

*Corresponding author

Those variables whose flips can bring a big benefit have a chance to be selected on the the second level, even if they do not satisfy the CC criterion.

The CCA heuristic is used to develop a new algorithm called Swcca (Smoothed Weighting and Configuration Checking with Aspiration). For showing the effectiveness of Swcca, we compare it with Sparrow2011 on a broad range of 3-SAT instances. Sparrow2011 is the improved version of the Sparrow algorithm which made a breakthrough on solving the random 3-SAT problem (Balint and Fröhlich 2010). It won the golden medal in the random satisfiable category of the SAT Competition 2011, and is considered to be the best local search solver for random 3-SAT instances. Our experiments show that the Swcca algorithm significantly outperforms Sparrow2011 on random 3-SAT instances.

Additionally, we compare Swcca with Sattime (Li and Li 2011) on structured instances from the crafted category of the SAT Competition 2011. Our experiments show that Swcca is competitive with Sattime on these structured instances. This is notable as Sattime was ranked 4th in the crafted category of the SAT Competition 2011, but only two portfolio solvers ppfolio (parallel and sequential versions) and sss did better than Sattime. Note that the good performance of the two ppfolio solvers are due to to the performance of TNM included in them (Roussel 2011). Sattime was developed from TNM and solved 109 instances, while the best CDCL solver only solved 93 instances in the category.

The remainder of this paper is organized as follows: some definitions and notations are given in the next section. Then we present the CC strategy and the CCA heuristic. After that, we describe the Swcca algorithm. Experimental results demonstrating the performance of Swcca are presented next. Finally we give some concluding remarks.

Definitions and Notations

Given a Conjunctive Normal Form (CNF) formula $F = C_1 \wedge C_2 \wedge ... \wedge C_m$ on a set of variables $\{x_1, x_2, ..., x_n\}$, the Boolean Satisfiability problem (SAT) consists in testing whether all clauses in F can be satisfied by some consistent assignment of truth values to variables. We use $V(F)$ to denote the set of all variables appear in the formula F. A *clause* is a disjunction of literals, where a *literal* is either a variable x or its negation $\overline{x}$. We say a literal l occurs in a clause, if this clause contains l. However, when we say a variable x occurs in a clause, we mean that this clause contains either x or $\overline{x}$. Two variables are neighbors iff they occur in at least one clause. Let $N(x) = \{y | y \in V(F)$ and y occurs in at least one clause with $x\}$, which is the set of all *neighboring variables* of variable x.

A (possibly partial) mapping $\alpha : V(F) \to \{0, 1\}$ is called an *assignment*. If α maps all variables to a Boolean value, it is called *complete*. For local search algorithms for SAT, a candidate solution is a complete assignment.

In a dynamic local search algorithm for SAT, a CNF formula F is combined with a weighting function w so that each clause $c \in F$ is associated with a positive integer number $w(c)$ as its weight. We use $cost(F, s)$ to denote the total weight of all unsatisfied clauses under an assignment s. For a variable x, let $score(x) = cost(F, s) - cost(F, s')$, measuring the benefit of flipping x, where s' is obtained from s by flipping x. A variable x is *decreasing* iff $score(x) > 0$. The *age* of a variable is defined as the number of search steps that have occurred since the variable was last flipped.

Configuration Checking

Originally introduced in (Cai, Su, and Sattar 2011), configuration checking (CC) is a 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 reduce cycles on the whole candidate solution.

The CC strategy is based on the concept *configuration*. In the context of SAT, 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 s the current assignment to $V(F)$, the $configuration$ of a variable $x \in V(F)$ is a vector C_x consisting of truth values of all variables in $N(x)$ under s (i.e., $C_x = s|_{N(x)}$, which is the assignment restricted to $N(x)$).

Given a CNF formula F, the CC strategy can be described as follows: When selecting a variable to flip, for a variable $x \in V(F)$, if the configuration of x has not been changed since x's last flip, which means the circumstance of x never changes, then it is forbidden to be flipped. 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 x's last flip; and $confChange[x] = 0$ on the contrary. During the search procedure, the variables with $confChange[x] = 0$ are forbidden to be flipped in the greedy mode, which could decrease blind unreasonable greedy search.

Previous SLS algorithms for SAT usually select the variable to flip based on properties of variables such as score (Hoos and Stützle 2000), break (Selman, Kautz, and Cohen 1994) and age (Gent and Walsh 1993); state-of-the-art SLS algorithms also utilize dynamic score (Tompkins and Hoos 2010) and functions combining different properties (Balint and Fröhlich 2010; Tompkins, Balint, and Hoos 2011). However, all these SLS algorithms neglect the circumstance of variables. Here the CC strategy takes into account the variables' circumstance information 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 information, but also its circumstance, such as the state of the community it belongs to.

Configuration Checking with Aspiration

Although the CC strategy shows its effectiveness in SLS algorithms for SAT, it is still in its infancy. We consider the CC strategy is too strict, as any variable whose configuration has not been changed since its last flip is forbidden to be flipped in the greedy mode, regardless of its score. This lack

of differentiation is a big disadvantage in our opinion. To overcome this drawback, we propose a new *pick-var* heuristic based on CC, which is called configuration checking with aspiration (CCA).

Before getting into the details of the CCA heuristic, we first give some definitions. A variable x is said configuration changed iff $confChange[x] = 1$. A *configuration changed decreasing* (CCD) variable is a variable with both $confChange[x] = 1$ and $score(x) > 0$. A *significant decreasing* (SD) variable is a variable with $score(x) > g$, where g is a positive integer large enough, and in this work g is set to the averaged clause weight (over all clauses) $\overline{w}$.

The CCA heuristic switches between the greedy mode and the diversification mode. In the greedy mode, there are two levels with descending priorities. On the first level it does a gradient walk, i.e., picking the CCD variable with the greatest score to flip. If there are no CCD variables, CCA activates the "aspiration criterion" to give a chance to the SD variables. Specifically, it selects the SD variable with the greatest score to flip if there is one, which corresponds to the second level. If there are neither CCD variables nor SD variables, CCA switches to the the diversification mode, where clause weights are updated, and the oldest variable in a uniformly random unsatisfied clause is picked to flip. In the CCA heuristic, all ties are broken by preferring the oldest variable.

The aspiration criterion selects the variables with great scores to flip. Note that in the diversification mode, an SLS algorithm with the CCA heuristic 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 criterion, such variables which are "mistakenly" flipped in the diversification mode will accumulate. This would delay the algorithm transferring to promising search areas. The aspiration criterion makes the algorithm correct such mistakes and thus can transfer to promising search areas in time.

The Swcca Algorithm

We use the CCA heuristic to develop a new SLS algorithm called Swcca. To focus on the essential part of the Swcca algorithm, we only present the pseudo code of its *pickVar* function, as the initialization is trivial: generating a random complete assignment s, initiating all clause weights as 1 and computing $scores$ of variables accordingly, and initiating all $confChange[x]$ as 1.

The CCA *pickVar*-function

The CCA *pickVar*-function is outlined in Algorithm 1, as described below:

The Swcca algorithm stores all CCD variables in a set *CCDVar* and all SD variables in a set *SDVar*. In each iteration, Swcca utilizes the CCA *pickVar*-function to pick a variable to flip in two modes: the greedy mode and the diversification mode.

The greedy mode: If *CCDVar* is not empty, the CCA function returns the variable with the greatest $score$ in the variable set *CCDVar*, breaking ties in favor of the oldest variable. If *CCDVar* is empty and *SDVar* is not empty, the CCA function returns the variable with the biggest $score$ in *SDVar*, breaking ties in favor of the oldest variable.

Algorithm 1: *pickVar*-function CCA

```
1 //greedy mode
2 if CCDVar≠ ∅ then return x ∈ CCDVar with the
  largest score, breaking ties in favor of the oldest one;
3 if SDVar≠ ∅ then return x ∈ SDVar with the largest
  score, breaking ties in favor of the oldest one;
4 //diversification mode
5 update clause weights;
6 pick a random unsatisfied clause c;
7 return the oldest variable in c;
```

The diversification mode: If both *CCDVar* and *SDVar* are empty, then the CCA function switches to the diversification mode. Specifically, it first updates clause weights: clause weights of all unsatisfied clauses are increased by one ; further, if the averaged weight $\overline{w}$ exceeds a threshold γ, all clause weights are smoothed as $w(c_i) := \lfloor \rho \cdot w(c_i) \rfloor + \lfloor (1 - \rho)\overline{w} \rfloor$. Then, it picks a uniform random unsatisfied clause, and returns the oldest variable in that clause.

Implementation of the CCA Heuristic

The complexity of the CCA heuristic depends on the implementation of the data structures: the confChange array, the SDVar and CCDVar sets. We describe our implementation below.

The *confChange* array is initialized by setting all $confChange[x]$ to 1. After that, when flipping a variable x, $confChange[x]$ is reset to 0, and for each $y \in N(x)$, $confChange[y]$ is set to 1.

The *SDVar* set is identified by checking all variables in unsatisfied clauses, as a variable not in unsatisfied clauses is impossible to be decreasing. We do not use more "clever" implementations for maintaining the *SDVar* set because it is not used that frequently, compared to the *CCDVar* set, which can be seen from the CCA *pickVar*-function.

The most important data structure of Swcca is the *CCDVar* set, which consists of all CCD variables. To maintain the *CCDVar* set, we employ a stack also named *CCDVar*; moreover, we utilize an auxiliary array $recorded$, whose element is an indicator — $recorded[x] = 1$ means x is stored in *CCDVar*; and $recorded[x] = 0$ on the contrary. At the end of the initialization stage of the Swcca algorithm, all variables with $dscore(x) > 0$ 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 frequently update the *CCDVar* stack, and along with updating *CCDVar*, the $recorded$ array 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. There are two cases in which the *CCDVar* stack is updated : (a) flipping a variable or (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-stage process. First, those variables that no longer satisfy the conditions ($dscore(x) > 0$ and $confChange[x] = 1$) are removed from *CCDVar*; second, those variables not in *CCDVar* but will satisfy the conditions after the flip are pushed into *CCDVar*. In the "removing" stage, all we do is scanning the *CCDVar* stack and removing those variables whose scores are no longer positive. We exclaim that by doing so we remove all the variables that no longer satisfy the conditions out of *CCDVar*. For the other condition $confChange[x] = 1$, in each step the only variable whose $confChange$ value changes from 1 to 0 is the flipped variable. If the flipped variable x was in *CCDVar* before the flip, then $score(x) < 0$ holds after flipping x (flipping x would make $score(x)$ to be its opposite number), so 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 an unsatisfied 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.

Experimental Results

In this section, we first present a brief introduction to the benchmarks we adopted, and describe some preliminaries about our experiments. Then, we divide the experiments into four parts. Part A is to compare Swcca with TNM (the winner of the satisfiable random category of SAT 2009 Competition), Sparrow2011 and Swcc on large 3-SAT instances from SAT 2009 Competition; part B is to compare Swcca with Sparrow2011 on 3-SAT instances from SAT 2011 Competition; and part C is to compare Swcca with Sparrow2011 on huge random 3-SAT instances with up to 80000 variables. While the first three parts of experiments are carried out on random 3-SAT instances, part D is to compare Swcca with Sattime on structured instances.

The Benchmarks

We evaluate the Swcca algorithm on four benchmarks. The first one contains all 3-SAT instances in the random large category of the SAT 2009 Competition ($2000 \leqslant \#var \leqslant 18000$), as well as all instances in the additional benchmark of the same category ($20000 \leqslant \#var \leqslant 26000$). The second one contains all 3-SAT instances in the random large category from the SAT 2011 Competition ($2500 \leqslant \#var \leqslant 50000$). The instances from random median category are too easy for a modern SLS solver that they are not included in our experiments. For these two benchmarks, there are 10 instances for each size. The clause-to-variable ratio is 4.2 for all instances.

As for the third benchmark, we generate 600 satisfiable huge random 3-SAT instances with a clause-to-variable ratio of 4.2 (the hardest ratio from the SAT competition used for such instances), according to the fixed clause length diversification model (no tautologies, no duplicate clauses, no duplicate literals in a clause). Their sizes range from 55000 variables to 80000 variables in increments of 5000 (100 instances each).

The fourth benchmark contains all satisfiable instances from the selected benchmark of the crafted category in the SAT Competition 2011[1]. We do not consider those instances labeled as unsatisfiable instances.

Experimental Setup

Swcca is implemented in C++ and compiled by g++ with the '-O2' option. In all experiments, we set $\gamma = 300$ and $\rho = 0.3$ for the smoothed clause weighting scheme in Swcca. The solvers TNM, Sparrow2011 and Sattime we use for comparison are the ones submitted to the SAT 2009 and 2011 competitions respectively. Swcc is the one tested in (Cai and Su 2011).

All experiments were run on a machine with an Intel Core E8400 with 3 GHz CPU and 3GB RAM under Linux. Each run terminates upon either finding a solution or reaching a given cutoff time which is set to 1000 seconds for the first two benchmarks and 1800 seconds (half an hour) for the last two benchmarks.

For the first two benchmarks, we run each solver 100 times for each instance and thus 1000 times for each class. For the third benchmark, we run each solver 5 times for each instance and thus 500 times for each class. For the fourth benchmark, we run each solver 10 times for each instance. We say an instance is solved by a solver if the solver finds a solution satisfying all clauses of the instance. For each solver on each class of instances, we report the success rate (the number of successful runs divided by the number of total runs), as well as the mean values of the run time (in seconds) and the number of flips.

Results

Part A: Table 1 presents the comparative performance results on the random 3-SAT benchmark from the SAT Competition 2009. Seen from the results, the performance of TNM and Swcc are not as good as those of the other two solvers. Swcca outperforms Sparrow2011 on all groups of instances in terms of both success rate and run time, and performs significantly better on large instances with $\#var \geq 20000$. In particular, altogether there is only one instance on which Swcca does not find a solution within the time limit in some runs, where it succeeds to find a solution in 94 runs. However, this is also the most difficult instance for Sparrow2011 and it succeeds in only 48 runs. Also, seen from Table 1, Swcca is about 3 to 4 times faster than Swcc, which indicates the effectiveness of the CCA heuristic.

Part B: We compare Swcca with Sparrow2011 with all (satisfiable) random 3-SAT instances from the SAT Compe-

[1] http://www.cril.univ-artois.fr/SAT11/bench/SAT11-Competition-SelectedBenchmarks.tar

Instance Class	TNM			Sparrow2011			Swcc			Swcca		
	suc	time	#flips(10^6)	suc	time	#flips(10^6)	suc	time	#flips(10^6)	suc	time	#flips(10^6)
3SAT-2000vars	100%	1.6	3.3	100%	1.4	2.1	100%	1.4	2.5	100%	0.8	1.5
3SAT-4000vars	94.9%	74	134.0	98.5%	48	70.6	100%	21.6	30.9	100%	10	15.1
3SAT-6000vars	99.5%	42	65.3	100%	17	21.2	100%	43	54.5	100%	14	18.1
3SAT-8000vars	98.5%	63	96.1	99.6%	34	39.1	99.1%	56	66.2	100%	20	22.2
3SAT-10000vars	99.8%	64	91.5	100%	14	15.7	100%	29	28.7	100%	13	13.7
3SAT-12000vars	97.1%	194	292.4	100%	46	49.4	100%	93	76.7	100%	31	30.4
3SAT-14000vars	94.2%	291	394.8	100%	49	50.3	100%	109	80.7	100%	38	34.7
3SAT-16000vars	96.5%	232	280.0	100%	29	29.5	100%	62	40.8	100%	23	20.5
3SAT-18000vars	92.3%	351	369.9	100%	37	36.2	100%	81	48.2	100%	30	25.3
3SAT-20000vars	45.6%	827	900.2	94.1%	236	183.5	92%	326	187.6	99.4%	131	97.7
3SAT-22000vars	59.8%	735	710.7	99.8%	108	97.8	99.3%	207	104.0	100%	65	41.4
3SAT-24000vars	59.0%	707	660.0	96.3%	158	134.7	93.5%	215	102.0	100%	83	52.2
3SAT-26000vars	55.8%	686	598.3	99.6%	106	89.1	97.5%	195	87.8	100%	66	35.9

Table 1: Comparative performance results on the large random 3-SAT instances from the SAT Competition 2009

tition 2001. Table 2 presents the comparative performance results of Swcca and Sparrow2011 on the random *large* 3-SAT benchmark from the SAT Competition 2001. For these large instances, Swcca also shows superiority to Sparrow2011, in terms of both success rate and run time. Particularly, altogether there is only one instance on which Swcca does not achieve a 100% success rate, where it achieves a 99% success rate. The obvious gap of success rate between the two solvers on the largest group (#var=50000) indicates a significant performance gap between Sparrow2011 and Swcca on difficult large random 3-SAT instances.

Swcc performs significantly worse than Swcca, especially on large instances. For example, on the group of instances with 50000 variables, it only achieves a success rate less than 50%. We do not report the detailed results of Swcc on this benchmark due to the limitation of space.

Also, to study how much the CCA strategy contribute to the Swcca algorithm, we run Swcca without the CCA strategy on this benchmark. The alternative algorithm with the CCA strategy fails to find a solution for all instances with $\#var \geqslant 5000$. This indicates that the CCA strategy plays a key role in the Swcca algorithm.

Part C: Table 3 presents the comparative performance results on huge 3-SAT instances, which shows that Swcca significantly outperforms Sparrow2011 on these huge instances. The success rates of Swcca on all groups of these huge instances are 100% or almost 100%, while the success rates of Sparrow2011 varies from 68% to 94.8%. The average run time of Swcca is less than half of that of Sparrow2011 on all groups of instances, except for the group with 80000 variables, which is nearly half of that of Sparrow2011. We are confident that Swcca would be able to solve even larger instances.

We would like to point out that the gap of step performance between Sparrow2011 and Swcca is much more obvious than the run time performance gap. We believe by optimizing the implementation, we can speed up Swcca. It is also worthy to note that the most significant idea in Sparrow2011 is the probability distribution technique (Balint and Fröhlich 2010) that used in the diversification mode, while the CCA strategy works in the greedy mode. We believe that these two significant techniques would cooperate well and result in a better local search SAT solver.

Instance Class	Sparrow2011			Swcca		
	suc	time	#flips(10^6)	suc	time	#flips(10^6)
3SAT-2500vars	99.5%	30	40.3	100%	13	22.3
3SAT-5000vars	100%	19	24.1	100%	14	19.2
3SAT-10000vars	99.9%	40	42.9	100%	26	26.5
3SAT-15000vars	100%	57	57.0	100%	41	37.0
3SAT-20000vars	99.9%	98	91.0	100%	62	50.1
3SAT-25000vars	98.9%	169	143.6	100%	89	65.2
3SAT-30000vars	97.9%	208	162.0	100%	106	72.1
3SAT-35000vars	92.4%	360	258.1	100%	186	95.6
3SAT-40000vars	90.7%	321	216.1	99.9%	162	76.8
3SAT-50000vars	67.8%	584	347.9	100%	262	119.6

Table 2: Comparative performance results of Swcca and Sparrow2011 on the random large 3-SAT instances from the SAT Competition 2011

Part D: We compare Swcca with Sattime on all satisfiable instances from the selected benchmark of the crafted category in the SAT Competition 2011. Sattime was the first SLS solver entering the last phase in the crafted category of a SAT competition and easily beating there all the CDCL algorithms, which have been considered more effective than local search for structured SAT problems for a longtime. Sattime was ranked 4th in the category, but only two portfolio solvers ppfolio (parallel and sequential versions) and sss did better than Sattime, thanks to the performance of TNM included in them. We also note that Sattime utilizes some reasonings before the local search procedure, which is helpful for solving structured instances.

Instance Class	Sparrow2011			Swcca		
	suc	time	#flips(10^6)	suc	time	#flips(10^6)
3SAT-55000vars	94.8%	591	380.8	100%	207	98.6
3SAT-60000vars	88%	730	456.6	100%	248	112.6
3SAT-65000vars	87.8%	843	511.6	100%	294	128.7
3SAT-70000vars	85.6%	880	516.3	99%	401	157.4
3SAT-75000vars	74.6%	1078	620.1	99.4%	477	189.1
3SAT-80000vars	68%	1187	658.1	97.2%	631	242.0

Table 3: Comparative performance results of Swcca and Sparrow2011 on huge random 3-SAT instances

Instance Class	#instances	Sattime		Swcca	
		suc	time	suc	time
289	15	100%	<0.01	100%	<0.01
automata-synchronization	7	0%	n/a	**17.5%**	**1498**
battleship	14	98.6%	26	**100%**	**<0.01**
GreenTao	3	**100%**	**37**	86.7%	91
sgen	10	**96%**	**199**	35%	1041
SRHD-SGI	28	**52.8%**	**930**	47.1%	1059
VanDerWaeden_pd_3k	7	75.7%	646	**100%**	**134**

Table 4: Comparative performance results of Swcca and Sattime on the selected crafted benchmark of the SAT Competition 2011. The results in bold indicate the best performance for a class of instances.

The results in Table 4 show that Swcca is competitive with Sattime on these structured instances. Particularly, while Sattime fails on all automata-synchronization instances, Swcca solves three of them. To the best of our knowledge, this is the first time that a local search solver solves such automata-synchronization instances.

We also compare Swcca and sattime on the 12 crafted forced satisfiable rbsat instances from the SAT Competition 2011. These rbsat instances are generated in the phase transition area according to the model RB (Xu and Li 2000; Xu et al. 2007). The experimental results show that the two solvers are competitive on these rbsat instances: both solvers solve the rbsat instances with less than 1000 variables in less than 50 seconds, but fail to find a solution on those with more than 1500 variables.

Related Work and Discussion

Heuristics in SLS algorithms for SAT can be divided into three categories: GSAT, WalkSAT and dynamic local search (DLS). Although there are considerable works on integrating CDCL heuristics in SLS algorithms for SAT (Li, Stallmann, and Brglez 2003; Hirsch and Kojevnikov 2005; Balint, Henn, and Gableske 2009; Audemard et al. 2010), researches towards improving SLS solvers for SAT are mainly about improving GSAT-like heuristics, or WalkSAT-like ones, or clause weighting techniques.

There are two strategies widely used in GSAT-like heuristics: the tabu mechanism (Mazure, Sais, and Grégoire 1997) and the promising decreasing variable (PDV) exploitation strategy proposed in G^2WSAT (Li and Huang 2005). It is worthy to note that all awarded SLS solvers in SAT competitions since 2007 follow G^2WSAT and switch between the greedy and diversification modes depending on the existence or not of promising variables (in our knowledge). However, Swcca switches between the two modes according the existence of the CCD and SD variables.

Remark that promising decreasing variables are a strict subset of CCD variables. For a variable to be CCD, it suffices that a neighbor is flipped and that the score is positive. To be promising, one or several neighbors should be flipped to make its score positive. When an increasing variable is flipped, it is CCD as soon as one of its neighbors is flipped and its score remains positive. However, it cannot be promising until its neighbors are flipped to make its score non-positive and then positive. The constraint to be promising is much stronger. In some sense, CCD and promising may be two extremities, there should be an intermediate notion more effective to investigate in the future.

Compared to GSAT-like heuristics, more works have been devoted to improving WalkSAT-like heuristics, most of which belong to the Novelty family, including Novelty and R-Novelty (McAllester, Selman, and Kautz 1997), Novelty+, AdaptNovelty+ (Hoos 2002), Novelty++ (Li and Huang 2005), Novelty+p (Li, Wei, and Zhang 2007) and so on. The most significant improvement recently is the selection mechanism based on probability distribution in Sparrow (Balint and Fröhlich 2010), which makes a break-through in solving the random 3-SAT problem.

A different line of research from GSAT and WalkSAT series is DLS algorithms using clause weighting techniques. The basic concept of clause weighting is to increase weight of unsatisfied clauses in local minima and thus help the search to avoid local minima. Well-known DLS algorithms include ESG (Schuurmans, Southey, and Holte 2001), SDF (Schuurmans and Southey 2001) and SPAS (Hutter, Tompkins, and Hoos 2002), DLM (Wu and Wah 2000) and PAWS (Thornton et al. 2004), as well as DDWF (Ishtaiwi et al. 2005). For a review on DLS algorithms, we refer to (Thornton 2005).

As with most current best solvers, Swcca combines heuristics from the three categories: GSAT-like, WalkSAT-like and DLS. The CCA heuristic is used to improve the GSAT-like heuristic (the greedy mode). In order to demonstrate the effectiveness of the CCA heuristic clearly, we keep the WalkSAT-like heuristic and the clause weighting technique in Swcca rather simple: the WalkSAT-like heuristic in the diversification mode of Swcca is a simplified version of Novelty, and the clause weighting scheme in Swcca is a simplified version of ESG.

Conclusions and Future Work

Inspired by the success of the configuration checking (CC) strategy on the Minimum Vertex Cover problem, we proposed a new variable selection heuristic called configuration checking with aspiration (CCA) for SLS algorithms for SAT. The CCA heuristic works on two levels in the greedy mode,

which is more flexible compared to the CC strategy. Moreover, the CCA heuristic has only one parameter for controlling when to activate the aspiration criterion, which is quite stable and do not need to be tuned manually.

We utilized the CCA heuristic to develop a new SLS algorithm called Swcca. The experiments show that Swcca is substantially faster for random 3-SAT than Sparrow2011, the winner of the 2011 SAT competition in the random category, and is competitive with Sattime for crafted instances, the best local search solver for crafted instances in the 2011 SAT competition. These results are exciting as Sparrow2011 represents the last breakthrough in solving hard random 3-SAT, and Sattime was the first SLS solver entering the last phase in the crafted category of a SAT competition and easily beating there all the CDCL algorithms, which have been considered more effective than local search for structured SAT problems for a longtime.

As for future work, we would like to design more sophisticated "configuration checking" techniques to further improve the state of the art in SLS algorithms for SAT. Also we would like to apply the CCA heuristic to other combinatorial search problems.

Acknowledgement

This work is partially supported by 973 Program (2010CB328103), ARC grants FT0991785 and DP120102489, and the Open Fund of the State Key Laboratory of Software Development Environment. We would like to thank the anonymous referees for their helpful comments.

References

Audemard, G.; Lagniez, J.-M.; Mazure, B.; and Sais, L. 2010. Boosting local search thanks to CDCL. In *Proc. of LPAR-10*, 474–488.

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

Balint, A.; Henn, M.; and Gableske, O. 2009. A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem. In *Proc. of SAT-09*, 284–297.

Cai, S., and Su, K. 2011. Local search with configuration checking for SAT. In *Proc. of ICTAI-11*, 59–66.

Cai, S.; Su, K.; and Chen, Q. 2010. EWLS: A new local search for minimum vertex cover. In *Proc. of AAAI-10*, 45–50.

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

Gableske, O., and Heule, M. 2011. Eagleup: Solving random 3-SAT using SLS with unit propagation. In *Proc. of SAT-11*, 367–368.

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

Glover, F. 1990. Tabu search – part ii. *ORSA Journal on Computing* 2(1):4–32.

Hirsch, E. A., and Kojevnikov, A. 2005. Unitwalk: A new SAT solver that uses local search guided by unit clause elimination. *Ann. Math. Artif. Intell.* 43(1):91–111.

Hoos, H. H., and Stützle, T. 2000. Local search algorithms for SAT: An empirical evaluation. *J. Autom. Reasoning* 24(4):421–481.

Hoos, H. H. 2002. An adaptive noise mechanism for WalkSAT. In *Proc. of AAAI-02*, 655–660.

Hutter, F.; Tompkins, D. A. D.; and Hoos, H. H. 2002. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proc. of CP-02*, 233–248.

Ishtaiwi, A.; Thornton, J.; Sattar, A.; and Pham, D. N. 2005. Neighbourhood clause weight redistribution in local search for SAT. In *Proc. of CP-05*, 772–776.

Kautz, H. A.; Sabharwal, A.; and Selman, B. 2009. Incomplete algorithms. In *Handbook of Satisfiability*. IOS Press. 185–203.

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

Li, C., and Li, Y. 2011. *Flippling time versus Satisfying in Local Search for SAT*. http://www.satcompetition.org.

Li, X. Y.; Stallmann, M. F. M.; and Brglez, F. 2003. A local search SAT solver using an effective switching strategy and an efficient unit propagation. In *SAT*, 53–68.

Li, C. M.; Wei, W.; and Zhang, H. 2007. Combining adaptive noise and look-ahead in local search for SAT. In *Proc. of SAT-07*, 121–133.

Mazure, B.; Sais, L.; and Grégoire, É. 1997. Tabu search for SAT. In *Proc. of AAAI-97*, 281–285.

McAllester, D. A.; Selman, B.; and Kautz, H. A. 1997. Evidence for invariants in local search. In *Proc. of AAAI-97*, 321–326.

Michiels, W.; Aarts, E. H. L.; and Korst, J. H. M. 2007. *Theoretical aspects of local search*. Springer.

Roussel, O. 2011. *Description of ppfolio*. http://www.cril.univ-artois.fr/SAT11/phase2.pdf.

Salhi, S. 2002. Defining tabu list size and aspiration criterion within tabu search methods. *Computers & OR* 29(1):67–86.

Schuurmans, D., and Southey, F. 2001. Local search characteristics of incomplete SAT procedures. *Artif. Intell.* 132(2):121–150.

Schuurmans, D.; Southey, F.; and Holte, R. C. 2001. The exponentiated subgradient algorithm for heuristic boolean programming. In *Proc. of IJCAI-01*, 334–341.

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

Thornton, J.; Pham, D. N.; Bain, S.; and Jr., V. F. 2004. Additive versus multiplicative clause weighting for SAT. In *Proc. of AAAI-04*, 191–196.

Thornton, J. 2005. Clause weighting local search for SAT. *J. Autom. Reasoning* 35(1-3):97–142.

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

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

Wu, Z., and Wah, B. W. 2000. An efficient global-search strategy in discrete lagrangian methods for solving hard satisfiability problems. In *Proc. of AAAI/IAAI-00*, 310–315.

Xu, K., and Li, W. 2000. Exact phase transitions in random constraint satisfaction problems. *J. Artif. Intell. Res. (JAIR)* 12:93–103.

Xu, K.; Boussemart, F.; Hemery, F.; and Lecoutre, C. 2007. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artif. Intell.* 171(8-9):514–534.