# An Efficient Implementation for WalkSAT

**Sixue Liu**
Institute for Interdisciplinary Information Sciences
Tsinghua University
Beijing, China
sixueliu@gmail.com

## Abstract

Stochastic local search (SLS) algorithms have exhibited great effectiveness in finding models of random instances of the Boolean satisfiability problem (SAT). As one of the most widely known and used SLS algorithm, WalkSAT plays a key role in the evolutions of SLS for SAT, and also hold state-of-the-art performance on random instances. This work proposes a novel implementation for WalkSAT which decreases the redundant calculations leading to a dramatically speeding up, thus dominates the latest version of WalkSAT including its advanced variants.

## Introduction

The Satisfiability problem (SAT) is one of the most well known NP-complete problem, which is of great importance to both theoretical and practical area. Given a propositional formula in conjunctive normal form(CNF), the task of the SAT problem is to find a Boolean assignment to the variables such that all clauses have true values. The $k$-SAT problem means that in the CNF, the number of variables in each clause is at most $k$.

The SAT problem has been well studied in last two decades and comes with numerous solvers and methods. The stochastic local search (SLS) is a widely used method, whose idea is to select the next flipping variable and change its value under the current assignment, until all the clauses are satisfied. Mainly, SLS consists of following two ways: greedy search and focused random walk, where the greedy search is to approach the optimal rapidly and the focused random walk is easier to jump out of a local optimal (Kautz *et al.* 2009). Most SLS solver are the combination of these two ways. The 3-SAT problem has been most studied, due to its simple form and statistical property. For $k$-SAT problem with long clauses, progress has been made in last few years (Cai and Su 2013) (Cai *et al.* 2013), but still can not catch up with the 3-SAT problem. Some algorithms even use different scheme to solve the 3-SAT problem comparing to those for $k$-SAT with long clause (Balint and Schöning 2012).

A interesting phenomenon in random $k$-SAT problem called phase transition has both empirical and theoretical meanings (Selman 1995) (Xu *et al.* 2012). Recent SLS algorithms can solve random 3-SAT problem near the phase transition point with variables up to millions (Luo *et al.* 2013). Although progress has been made at the phase transition point (Luo *et al.* 2014), the variables number of solved instances within rational time never reached to 20 thousand.

This work is devoted to more efficient SLS algorithm based on focused random walk framework for random $k$-SAT at the phase transition point. First, we exploit the 0-break variable in the variable picking heuristic. Second, we propose a new scheme called separating-non-caching to compute the $break$ value, which decreases some unnecessary calculations and improves the efficiency. Combining all these, we design a new SAT solver dubbed WalkSNC (WalkSAT with separated-non-caching).

The experimental results show that WalkSNC significantly outperforms the latest version of WalkSAT including its state-of-the-art variants, especially on large scale benchmarks. SAT Competition has been held for more than 10 years to evaluate state-of-the-art SAT solvers. Our benchmark includes random $k$-SAT instances on the phase transition point from SAT Competition 2013 and 2014, and many larger instances generated by the uniform random $k$-SAT generator.

The rest of this paper is organized as follows. Some necessary notations and definitions are given in the next section. In section 2 we show the importance of the 0-break variable. Then the separating-non-caching technology is introduced. And experimental evaluations are illustrated after that. Finally, we give conclusions of this work and future directions.

## Preliminaries

Given a Conjunctive Normal Form(CNF) formula $F = c_1 \bigwedge ... \bigwedge c_m$ on a variables' set $V = \{v_1, v_2, ..., v_n\}$, where $c_i$ is a clause and consists of literals: boolean variables or their negations. A $k$-SAT formula is a CNF where each clause contains at most $k$ literals. The $ratio$ of a CNF is defined as the ratio of the the number of clauses and the number of variables. An assignment $\alpha$ is called complete if it matches every variable with TRUE or FALSE. We say a literal is a true literal if it evaluates to TRUE under $\alpha$. The task of the SAT problem is to answer whether there exists a complete assignment such that all clauses are satisfied.

SLS algorithms under focused random walk framework first choose an unsatisfied clause $c$, then choose a flip variable from $c$ according to some rules (Algorithm 1). These

**Algorithm 1:** Focused Random Walk Framework

**Input**: CNF-formula $F$, $maxSteps$
**Output**: A satisfying assignment $\alpha$ of $F$, or $Unknown$
**begin**
  $\alpha \leftarrow$ random generated assignment;
  **for** $step \leftarrow 1$ **to** $maxSteps$ **do**
    **if** $\alpha$ $satisfies$ $F$ **then return** $\alpha$;
    $c \leftarrow$ an unsatisfied clause chosen randomly;
    $v \leftarrow pickVar(c)$
    $\alpha \leftarrow \alpha$ with $v$ flipped;
  **return** $Unknown$

rules are usually based on variables' information like $break$ and $make$. The $break$ value of $v$ is the number of clauses which will become unsatisfied from satisfied after flipping $v$. While the $make$ value is the number of clauses which will become satisfied from unsatisfied after flipping $v$. A traditional SLS algorithm called WalkSAT/SKC uses a simple rule to pick variable: if there exists a variable with $break = 0$, flip it, otherwise flip a random variable with probability $p$, or a variable with minimal $break$ with probability $1 - p$. Another SLS algorithm called probSAT also uses $break$ value only, but in a completely probabilistic way. Some recent SLS algorithms also utilize some other information of variables to obtain more complex rules (Cai and Su 2012): the $neighborhood$ of a variable $v$ are all the variables that occur in at least one same clause with $v$. The $score$ of a variable is defined as the sum of weights of clauses (at least 1) which will become satisfied from unsatisfied after flipping that variable. If variable $v$'s neighborhood has been flipped since $v$'s last flip, $v$ is called $configuration\ changed$ variable, and $configuration\ change\ decreasing$(CCD) variables if $score(v) > 0$ too. This notion has a significant influence to state-of-the-art SLS algorithms, and we will illustrate the connection between our algorithm and it.

## The 0-Break Variable

In this section, we define 0-break variable and show its importance.

**Definition 1.** *0-break variable is the variable whose break value is 0 in an unsatisfied clause.*

The $configuration\ changed\ decreasing(CCD)$ variables play a key role in $configuration\ checking$ related algorithms, they are variables whose neighborhoods have be flipped since their last flipping($configuration\ changed$) and their $score$ are positive($decreasing$). Let's illustrate the connection between 0-break and $CCD$ variable.

**Lemma 1.** *The 0-break variable $v$ is also a $CCD$ variable.*

*Proof.* If none of $v$'s neighborhood has been flipped since $v$'s last flipping, $break(v)$ remains the same from that time. So $break(v)$ was also 0 when it was just flipped. But it is impossible because it was picked from an unsatisfied clause, flipping $v$ would turn that clause back to unsatisfied again, which means $break(v)$ was at least 1. So $v$'s configuration must have been changed. Noticing that the $score$ of a variable defined in $configuration\ checking$ related algorithms is the sum of weights of clauses(at least 1) which will become satisfied from unsatisfied after flipping that variable, so it is no less than $make - break$. However, for 0-break variable, $make$ value is at least 1 because it's in an unsatisfied clause, and $break$ value is 0. So its $score$ must be positive. Therefore, we have the conclusion that 0-break variable is also a $CCD$ variable. □

**Algorithm 2:** The $pickVar$ Function of Generalized WalkSAT

**Input**: An unsatisfied clause $c$
**Output**: A variable $v \in c$
**begin**
  **if** $\exists$ variable $x \in c$ with $break(x) = 0$ **then**
    With probability $p_0$:
      $v \leftarrow$ a variable in $c$ chosen at random;
    With probability $1 - p_0$:
      $v \leftarrow x$;
  **else**
    With probability $p_1$:
      $v \leftarrow$ a variable in $c$ chosen at random;
    With probability $1 - p_1$:
      $v \leftarrow$ a variable in $c$ with minimum $break$;
  **return** $v$

| Solvers | W0 | W1 | W2 | W3 | W4 | W5 |
|---|---|---|---|---|---|---|
| $p_0$ | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.492 |
| $p_1$ | 0.567 | 0.556 | 0.542 | 0.527 | 0.510 | 0.492 |

Table 1: Parameters of generalized WalkSAT for 3-SAT

Noticing that the reverse of Lemma 1 is not necessarily true.

To show how 0-break variable affects focused random walk based algorithm, we design a generalized version of WalkSAT, outlined in Algorithm 2, as described below. First we check if there exists 0-break variable. If does, a random variable is returned with probability $p_0$, or return the the variable with minimal $break$ value(0-break variable) with probability $1 - p_0$. Otherwise, we follow the same process with another probability $p_1$.

Intuitively, $p_0$ should be less than $p_1$. In fact, $p_0 = 0$ and $p_0 = p_1$ are two prototypes of WalkSAT (McAllester *et al.* 1997). We report the best $p_1$ corresponding to each $p_0$ in Table 1, $p_0$ ranges from 0 to $p_1$, as well as their performances on 3-SAT in Figure 1.

We use random 3-SAT instances from SAT Challenge 2012 with $ratio = 4.2$. The 'org' line represents the original caching implementation with XOR technology, while the 'snc' line is under separating-non-caching implementation introduced later with some special modification for WalkSAT.

Noticing that W5 benefits the most from our new implementation, because no calculation is needed with probability 0.492 when picking a variable. But it still can not compete

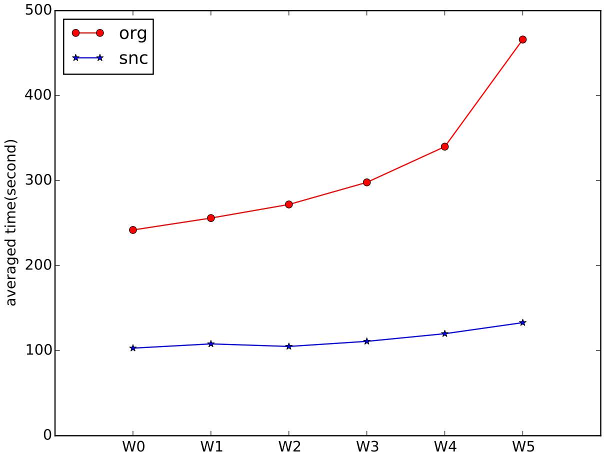


Figure 1: Performances under different parameters

W0. When $p_0$ becomes bigger, the algorithm becomes less efficient. When we come to the original one, the preference is distinct, because the time complexity in each step is almost the same. As for random SAT with long claus, W0 also dominates others. As a result, 0-break variable bring profits to the search process and should be flipped with priority.

## Separating-non-caching Technology

Implementation affects the performances of SLS algorithm very much. The latest version of probSAT uses caching scheme with XOR technology (Balint *et al.* ), while WalkSAT in UBCSAT framework (Tompkins and Hoos 2005) and the latest version of WalkSATlm (Cai *et al.* 2014) are under non-caching implementation. In this section, we propose a more efficient implementation called separating-non-caching. The 'separating' term means separating the non-caching process of calculating $break$, to find 0-break variables as soon as possible to reduce unnecessary calculations.

Recall the caching scheme updates every information including the $break$ value of each variables. However, if there exists variable with $break = 0$, the other variable's $break$ value is useless. We try to reduce the wasting calculations and only compute what we need.

Under our new implementation, the $flip$ operation only updates the unsatisfied clauses' set and the true literal numbers of every clause, but leave the $break$ calculation to $pickVar$ function. There are some necessary definitions to compute $break$ value.

**Definition 2.** *For each clause c, $NT(c)$ donates the number of true literals in c.*

$NT(c) = 0$ means $c$ is unsatisfied, satisfied clause always has positive $NT$.

**Definition 3.** *For each variable v, $TLC(v)$ donates all the clauses containing the true literal $v$ if $v = TRUE$ under the current assignment or $\bar{v}$ vice versa.*

---

**Algorithm 3:** The $pickVar$ Function of separating-non-caching

---

**Input**: An unsatisfied clause $c$
**Output**: A variable $v \in c$

```
1  begin
2      Generate a random order of all the variables in c;
3      foreach v ∈ c do
4          Initiate all the clauses in TLC(v) as unvisited;
5          zero ← TRUE;
6          foreach ci ∈ TLC(v) do
7              mark ci as visited in TLC(v);
8              if NT(ci) = 1 then
9                  zero ← FALSE;
10                 break;
11         if zero=TRUE then
12             return v;
13     foreach v ∈ c do
14         break(v) ← 1;
15         foreach unvisited ci ∈ TLC(v) do
16             if NT(ci) = 1 then
17                 break(v) ← break(v) + 1;
18     Choose variable according to line 7 to line 12 in
       algorithm 2, with noise setting 0.567;
```

---

If $v$ is TRUE under the current assignment, all clauses contains positive $v$ become $TLC(v)$. Else if $v$ is FALSE, all the clauses contains negative $v$ are $TLC(v)$. All $c$ in $TLC(v)$ with only one true literal will contribute 1 to $break(v)$. Because $v$ is the only one true literal in $c$, flipping $v$ will falsify this literal and make $c$ unsatisfied.

We need an additional boolean flag $zero$ to donate whether 0-break variables exists. The implementation is outlined in Algorithm 4.

In the separated-non-caching implementation outlined in algorithm 3, if there are more than one 0-break variables, return a random one. So in line 2, we first generate a random order to guarantee the first variable with $break = 0$ is a random 0-break variable. That's why line 11 can directly return. Line 3 to line 12 is to decide whether exists 0-break variable. If not, return the a random variable with probability 0.567, or return the variable with minimal $break$ value with the remaining probability. Because we mark the clauses in $TLC$, so at most $|c| \times |TLC|$ clauses are visited. The average size of $TLC$ is $k \times ratio/2$, $k$ donate $k$-SAT. For random 3-SAT with $ratio = 4.2$, it's about $3 \times 3 \times 4.2/2 = 18.9$ clauses to visit. However, due to the existence of 0-break variable, the average visited clauses are much less than 18.9.

Table 2 shows the percentage of 0-break variables among all flipping variables. We can see that for 3-SAT and 4-SAT, about 10% of the flipping of WalkSAT only executes simple calculations to find 0-break variable, which improve the efficiency. However, for longer clause SAT, due to the few 0-break variable, the speeding up is not so distinct. Moreover, if 0-break variable doesn't exists, $k \times k \times ratio/2$ clauses

| 3-SAT | 4-SAT | 5-SAT | 6-SAT | 7-SAT |
|---|---|---|---|---|
| 9.8% | 10.7% | 7.9% | 6.5% | 4.3% |

Table 2: Percentage of 0-break variables among all flipping variables

| Implementations | 3-SAT | 4-SAT |
|---|---|---|
| Caching without XOR | 3.98 | 2.46 |
| XOR-caching | 4.99 | 2.95 |
| Non-caching | 4.84 | 2.78 |
| Separating-non-caching | **5.30** | **3.07** |

Table 3: Average $10^6$ flips per second

have to be visited to compute all the variables' $break$ value averagely. It's about 263 clauses for 5-SAT, even large up to 2150 for 7-SAT, which is too costly. That's why we don't recommend to use non-caching technology for 5,6,7-SAT. Some experiments results are exhibited on random $k$-SAT of SAT Challenge 2012, which indicates that our separating-non-caching technology can executes more flips within the given time, outperforms others on 3-SAT and 4-SAT.

## Experimental Evaluations

We carry out large-scale experiments to evaluate WalkSNC on random $k$-SAT instances at the phase transition point.

### The Benchmarks

We adopt 3 random random benchmarks from SAT competition 2013 and 2014 as well as 100 instances we generated randomly. The experiments for $k$-SAT ($k > 3$) are not reported here but will be shown in the full version.

- SC13: 50 different variables instances with $ratio = 4.267$. From the threshold benchmark of the random SAT track of SAT competition 2013[1].
- SC14: 30 different variables instances with $ratio = 4.267$. From the threshold benchmark of the random SAT track of SAT competition 2014[2].
- V-$10^5$: Generated by the SAT Challenge 2012 generator with 50 instances for 100,000 variables, $ratio = 4.2$.
- V-$10^6$: Generated by the SAT Challenge 2012 generator with 50 instances for 1,000,000 variables, $ratio = 4.2$.

Note that the instances from SAT Competition 2013 and 2014 have approximately half unsatisfied fraction.

### The Competitors

We compare WalkSNC with the latest version of WalkSAT downloaded from WalkSAT homepage [3], and a state-of-the-art implementation based on non-caching WalkSATlm, and its variant probSAT which is the championship of SAT competition 2013 random track.

### Evaluation Methodology

The cutoff time is set to 5000 seconds as same as in SAT Competition 2013 and 2014, which is enough to test the performance of SAT solvers. Each run terminates finding a satisfying within the cutoff time is a successful run. We run each solver 10 times for each instance from SAT Competition 2013 and 2014 and thus 500 runs for each class. We report "suc" as the ratio of successful runs and total runs, as well as the "par10" as the penalized average run time(a unsuccessful run is penalized as $10\times$ cutoff time). The result in **bold** indicates the best performance for a class.

All the experiments are carried out on our machine with Intel Core Xeon E5-2650 2.60GHz CPU and 32GB memory under Linux.

### Experimental Results

Table 4 shows the comparative results of WalkSNC and their state-of-the-art competitors on the 3-SAT threshold benchmark. The best performance is achieved by WalkSNC, the others performs relatively poor. Considering the constant speeding up, the average time over all the successful runs of WalkSATv51 is almost 1.5 times of WalkSNC, and WalkSATlm is also 25% slower than us. The comparison of data we report on par10 is even more distinct. We state that this improvement is more conspicuous than the average flips per second showed in table 3 is because other optimization incorporated in our solver like pointers using and memory management.

## Conclusions and Future Work

This work opens up a totally new research direction in improving SLS for SAT: instead of calculating and utilizing extra and precise information of variables to decide which one to be flipped, there are much more things to dig using the simple information. What matters most is balancing the cost of calculating them and the benefits they bring. Additionally, this new technology can be easily adapted to new algorithms based on WalkSAT and probSAT.

## References


Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 16–29. Springer, 2012.

Adrian Balint, Armin Biere, Andreas Fröhlich, and Uwe Schöning. Improving implementation of sls solvers for sat and new heuristics for k-sat with long clauses.

Shaowei Cai and Kaile Su. Configuration checking with aspiration in local search for sat. In *AAAI*, 2012.

Shaowei Cai and Kaile Su. Comprehensive score: Towards efficient local search for sat with long clauses. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 489–495. AAAI Press, 2013.


[1]http://www.satcompetition.org/2013/downloads.shtml/Random Benchmarks

[2]http://www.satcompetition.org/2014/downloads.shtml/Random Benchmarks

[3]https://www.cs.rochester.edu/u/kautz/walksat/

| Instance Class | WalkSATv51 suc par10 | WalkSATlm suc par10 | probSAT suc par10 | WalkSNC suc par10 |
|---|---|---|---|---|
| SC13 | 10.2% 45340 | 26.4% 36980 | 29.1% 35010 | **37.4%** **31198** |
| SC14 | 9.6% 45120 | 27.1% 36603 | 19.0% 40994 | **35.5%** **33010** |
| V-$10^5$ | 95.3% 3453 | 99.0% 1292 | 100% 763 | **100%** **432** |
| V-$10^6$ | 89.2% 6159 | 98.0% 2274 | 99.1% 1514 | **99.8%** **499** |

Table 4: Comparison on random 3-SAT


Shaowei Cai, Kaile Su, and Chuan Luo. Improving walksat for random k-satisfiability problem with $k > 3$. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

Shaowei Cai, Chuan Luo, and Kaile Su. Improving walksat by effective tie-breaking and efficient implementation. *The Computer Journal*, page bxu135, 2014.

Henry A Kautz, Ashish Sabharwal, and Bart Selman. Incomplete algorithms. *Handbook of Satisfiability*, 185:185–203, 2009.

Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su. Focused random walk with configuration checking and break minimum for satisfiability. In *Principles and Practice of Constraint Programming*, pages 481–496. Springer, 2013.

Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su. Double configuration checking in stochastic local search for satisfiability. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *AAAI/IAAI*, pages 321–326, 1997.

Bart Selman. Stochastic search and phase transitions: Ai meets physics. In *IJCAI (1)*, pages 998–1002, 1995.

Dave AD Tompkins and Holger H Hoos. Ubcsat: An implementation and experimentation environment for sls algorithms for sat and max-sat. In *Theory and Applications of Satisfiability Testing*, pages 306–320. Springer, 2005.

Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. Predicting satisfiability at the phase transition. In *AAAI*, 2012.