# Faster Implementation for WalkSAT

Shaowei Cai

*Queensland Research Lab, NICTA, Australia*

**Abstract**

The Boolean Satisfiability problem (SAT) is a prototypical NP-complete problem, and is central to many domains of computer science and artificial intelligence. One of the most famous local search algorithms for SAT is WalkSAT, which is an initial algorithm that has wide influence among modern local search algorithms. Especially, WalkSAT is shown to be very efficient in solving large random 3-SAT instances. The runtime of a local search algorithm is determined not only by its search behavior, but also the complexity of each iteration. This work proposes an efficient implementation for WalkSAT, which leads to twice speedup over the latest (and fastest) implementation version of WalkSAT, according to the experiments on random 3-SAT instances near the phase transition with up to 1 million variables.

*Keywords:* WalkSAT, Local Search, Algorithm Implementation

## 1. Introduction

The Boolean satisfiability (SAT) problem is a prototypical NP-complete problem, and is an important subject of study in many areas of computer science and artificial intelligence. Given a conjunctive normal form (CNF) formula, the SAT problem is to decide whether there is an assignment to its variables that satisfies the formula. Algorithms for solving SAT can be mainly categorized into two classes: complete algorithms and stochastic local search (SLS) algorithms.

Among SLS algorithms for SAT, WalkSAT [1] stands out as one of the most influential algorithms. Especially, it is still competitive with the sate of the art in solving large random 3-SAT instances [2, 3]. Recently, there has been increasing interest in WalkSAT due to the discovery of its great power on large random 3-SAT instances. In 2004, Aurell $et\ al.$ observed that WalkSAT was far more powerful than had been appreciated [4]. Further empirical studies showed that WalkSAT (with $p = 0.567$) scales linearly in $n$ for random 3-SAT instances with a $ratio$ up to 4.2 [2, 5]. The more recent study in [3] illustrated WalkSAT has extremely good performance on random 3-SAT instances ($r = 4.2$) with up to $5 \times 10^5$ variables.

One significant factor of the success of WalkSAT is its low complexity each step. Recently, the authors of WalkSAT proposed a new implementation (WalkSAT_v50),

*Email address:* shaoweicai.cs@gmail.com (Shaowei Cai)

which is optimized according to the suggestions by Donald E. Knuth and is 20% faster than earlier versions.

This work proposes a more efficient implementation for WalkSAT. As will be introduced in Section 3, WalkSAT mainly utilizes the property $break$ to pick a variable to flip from a falsified clause. Our implementation relies on the fact that for a variable $x$, a clause contributes (one) to $break(x)$ only when the clause has only one true literal, which also happens to be a true literal of $x$.

As usual, we record the number of true literals for each clause $c$. However, the key data structures are two index arrays for each variable $x$: one stores the index numbers of clauses where its positive literal $x$ appears, and the other stores those of clauses where its negative literal $\neg x$ appears.

Employing these data structures in a non-caching implementation framework, we can compute the $break$ values for variables in a clause very quickly. To compute $break(x)$, we only need to scan one of the two index arrays for $x$, i.e., the one that records index numbers of clauses containing true literals of $x$, which contains only $\frac{1}{2} \cdot \frac{km}{n} = \frac{k}{2} \cdot r$ elements for $k$-SAT with the clause-to-variable ratio $r$. For 3-SAT instances with $r = 4.2$ (near the phase transition), the expectation of this number is $\frac{3}{2} \cdot 4.2 = 6.3$.

**We would like to note that, although similar data structures have been used in DPLL algorithms (and maybe some SLS ones), this is the first time to combine them with the non-caching computing framework, to the best of our knowledge. It is this combination of data structure and non-caching computing framework that makes our implementation so efficient.**

The experiments show that our new implementation of WalkSAT is twice faster than WalkSAT_v50 on large random 3-SAT instances. When compared to the state-of-the-art local search solver probSAT on random 3-SAT instances, our implementation of WalkSAT is also obviously faster. The codes of our implementation of WalkSAT can be downloaded online[1].

The remainder of this paper is organized as follows. Some definitions and notations are given in the next section. Section 3 introduces the WalkSAT algorithm. Section 4 proposes an efficient implementation for WalkSAT. Experiments for testing the new implementation of WalkSAT are carried out in Section 5. Finally, we give some concluding remarks.

## 2. Definitions and Notations

Given a set of $n$ Boolean $variables$ $\{x_1, x_2, ..., x_n\}$, a $literal$ is either a variable $x$ (which is called positive literal) or its negation $\neg x$ (which is called negative literal). A $clause$ is a disjunction of literals. A Conjunctive Normal Form (CNF) formula $F = C_1 \wedge C_2 \wedge ... \wedge C_m$ is a conjunction of clauses. The Boolean Satisfiability problem (SAT) consists in testing whether all clauses in a given CNF formula $F$ can be satisfied by some consistent assignment of truth values to variables. The clause-to-variable ratio

[1] www.shaoweicai.net/SAT.html

of a CNF formula with $n$ variables and $m$ clauses is $r = m/n$. For a given clause $C$, the set of all variables that appear in $C$ is denoted as $Vars(C)$.

A well known generation model for SAT is the random $k$-SAT model [6]. A random $k$-SAT formula with $n$ variables and $m$ clauses, is a CNF formula where the clauses are chosen uniformly, independently and without replacement among all $2^k\binom{n}{k}$ non-trivial clauses of length $k$, i.e., clauses with $k$ distinct, non-complementary literals.

Given an assignment $\alpha$, if a literal evaluates to true under $\alpha$, we say it is a $true\ literal$; otherwise, we say it is a $false\ literal$. A clause is *satisfied* if it has at least one true literal, and *falsified* if it has no true literal. In this work, when talking about the states of literals and clauses, the assignment $\alpha$ is always the current assignment, and thus omitted. An important concept in WalkSAT is the variable property $break$. For a variable $x$, $break(x)$ is the number of satisfied clauses that would become falsified by flipping $x$.

## 3. The WalkSAT Algorithm

This section introduces the WalkSAT algorithm, which was proposed in [1]. WalkSAT is one of the most influential SLS algorithms for SAT. It provides basic algorithmic ideas in local search for SAT, and is still competitive with the state of the art in solving random 3-SAT instances.

In each step, WalkSAT picks a variable to flip as follows. First, a falsified clause $C$ is selected randomly. If there exist variables with a $break$ value of 0 in clause $C$, one of such variables is flipped, with ties broken randomly. If no such variable exists, then with a certain probability $p$ (the noise parameter), one of the variables from $C$ is randomly selected; in the remaining cases, one of the variables with the minimum $break$ value from $C$ is selected, with ties broken randomly. For details about the WalkSAT algorithm, please refer to [7] and [8].

## 4. More Efficient Implementation for WalkSAT

The runtime of a local search algorithm is determined not only by its search behavior, but also the complexity of each iteration. In this section, we present an efficient implementation for WalkSAT, which significantly improve the performance of WalkSAT.

### *4.1. The Compute-when-evaluate Framework*

For GSAT-family algorithms, the caching and incremental updating scheme (for calculating $scores$) is much more efficient than the one in which $scores$ are computed from scratch in every step. However, for the WalkSAT algorithm, computing a variable's $break$ value straight before its evaluation (for being selected or not) leads to better performance. For convenience, we refer to this implementation as "compute-when-evaluate", according to how it works. In this work, we adopt the compute-when-evaluate implementation framework.

Based on this compute-when-evaluate framework, each step of WalkSAT is described in Algorithm 1. Two important notations are $break_{min}$ and $minbVars$. For a clause $C$,

$$break_{min} = \min_{x \in Vars(C)} break(x) \tag{1}$$

$$minbVars = \{x | x \in Vars(C), break(x) = break_{min}\} \tag{2}$$

That is, $break_{min}$ is the minimum $break$ value among $break$ values of all variables in the selected falsified clause, and $minbVars$ is a stack that stores all variables with the minimum $break$ in the selected falsified clause.

**Algorithm 1**: Each Step of WalkSAT in Compute-when-evaluate Implementation Framework

```
C := a falsified clause chosen at random;
break_min := MAXINT, minbVars := ∅;
for i := 1 to |C| do
    v := i^th variable in clause C;
    compute break(v);
    if break(v) < break_min then
        break_min := break(v);
        reset minbVars so that v is its only element;
    else if break(v) = break_min then
        push v into minbVars;
if break_min = 0 then
    v := a random variable in minbVars;
else
    With probability p:
        v := a random variable in C;
    With probability 1 − p:
        v := a random variable in minbVars;
flip v;
```

### *4.2. Computing $break$ Efficiently*

The efficiency of implementations based on the compute-when-evaluate framework mainly depends on the efficiency of computing the break value of variables. In our implementation, we need the following data structure to compute the break values of variables:

1. $TrueLitCount$: an array that records the number of true literals for all clauses. For example, $TrueLitCount(0) = 2$ means the first clause (suppose the index numbers of clauses begin with 0) has 2 true literals;
2. $PosLitClause(x)$ for each variable $x$: an array that stores the index numbers of clauses where the positive literal $x$ appears. For example, $PosLitClause(x) = \{0, 3, 6, 8, 23, 90\}$ means the literal $x$ (only) appears in clauses whose index numbers are 0,3,6,8,23,90.

3. $NegLitClause(x)$ for each variable $x$: an array that stores the index numbers of clauses where the negative literal $\neg x$ appears.

Our implementation relies on the following observation.

**Observation 1.** *For a variable $x$, a clause contributes (one) to $break(x)$ only when the clause has only one true literal, which is a true literal of $x$.*

The procedure of computing $break(x)$ for a variable $x$ is given in Algorithm 2. As shown in Algorithm 2, to compute $break(x)$, **we only need to scan one of the two index arrays for x**, i.e., $PosLitClause(x)$ or $NegLitClause(x)$. For a variable $x$ that we would like to compute its break value, first we initialize $break(x)$ as 0. Then, if the truth value of $x$ is TRUE under current assignment, we scan the $PosLitClause(x)$ array and compute $break(x)$ as follows. For each $c \in PosLitClause(x)$, we check whether clause $c$ contains exactly one true literal. If this is the case, it means literal $x$ is the only true literal in clause $c$. Then, flipping the value of $x$ would make clause $c$ become falsified from satisfied, and thus $break(x)$ should be increased by one. If the truth value of $x$ is FALSE under current assignment, we scan $NegLitClause(x)$ and compute $break(x)$ similarly.

It is clear that the procedure in Algorithm 2 correctly computes $break(x)$, if $x$ has the minimum $break$ among all variables in the clause. For those variables whose $break$ values are greater than the current value of $break_{min}$, it is not necessary to compute their $break$ values, as we only care which are variables with the minimum $break$ in WalkSAT, rather than the concrete $break$ value of each variable.

This computing procedure is very efficient. Given a $k$-SAT instance, for a variable $x$, the size of either $PosLitClause(x)$ or $NegLitClause(x)$ is $\frac{1}{2} \cdot \frac{Km}{n} = \frac{k}{2}r$. Particularly, for 3-SAT instances, in order to compute the $break$ value for a variable, we only need to perform $\frac{3}{2}r$ iterations, which is approximately only 6 when $r = 4.2$. Moreover, as we just mentioned above, once the $break$ value of a variable is greater than $break_{min}$, we can immediately quite and return, as this variable cannot have the minimum $break$.

Previous implementations such as WalkSAT_50 also store clauses for positive literal and negative literal for each variable. However, this data structure — storing clause index numbers for positive literal and negative literal in two arrays, is new in local search. Especially, the power of this data structure is exploited fully in the compute-when-evaluate framework, and their combination allows us to significantly improve the performance of WalkSAT.

## 5. Experimental Results

We carry out experiments to compare our implementation of WalkSAT with the existing fastest implementation, as well as the state-of-the-art local search solver probSAT, on large random 3-SAT instances.

*Benchmarks*

For experiments, we generate 300 satisfiable huge random 3-SAT instances near the phase transition ($r = 4.2$) according to the random $k$-SAT model, which have been

**Algorithm 2**: Computing $break(x)$

```
1 break(x) := 0;
2 if the truth value of x is TRUE then
3     foreach clause c ∈ PosLitClause(x) do
4         if TrueLitCount(c) = 1 then break(x)++;
5         if break(x) > break_min then return;
6 else
7     foreach clause c ∈ NegLitClause(x) do
8         if TrueLitCount(c) = 1 then break(x)++;
9         if break(x) > break_min then return;
```

cited as the hardest group of SAT problems [9]. There are 100 instances for each size ($n = 10^5, 5 \times 10^5, 10^6$). Note that we do not include other kinds of benchmarks, such as random 5-SAT, 7-SAT and structured ones, because the WalkSAT algorithm is not good at solving those instances, and thus the implementation does not have an obvious impact on the performance.

*Solvers for Comparison*

We compare our implementation of WalkSAT with the latest version (v50) of WalkSAT from its author's website[2], with the noise parameter set to 0.567. Note that this version (v50) has done code optimizations suggested by Donald E. Knuth, and is 20% faster than earlier versions[3]. For the purpose of comparision with state-of-the-art solver on random 3-SAT, we also test probSAT [3] in our experiments, which made the recent progress in solving random 3-SAT and won the random track of SAT Competition 2013.

*Soft- and Hard-ware*

Our implementation of WalkSAT is programmed in C++, complied by g++ with the '-O3' option. The noise parameter is set to 0.567, as suggested in literature [2, 5].

All experiments are carried out on a workstation under Linux, using 2 cores of Intel(R) Core(TM) i7-2640M 2.8 GHz CPU and 8 GB RAM. The experiments are conducted with EDACC [10], an experimental platform for testing SAT solvers, which has been used for SAT Challenge 2012 and SAT Competition 2013. Each solver was performed on each instance once within a cutoff time of 10000 seconds so that each run can find a satisfying assignment within the cutoff time. We compare the averaged run time for each instance class.

[2] http://www.cs.rochester.edu/~kautz/walksat/Walksat_v50.zip

[3] http://www.cs.rochester.edu/~kautz/walksat/

| #var | WalkSAT_v50 | WalkSAT(this work) | probSAT |
|---|---|---|---|
| $10^5$ | 335 | **132** | 339 |
| $5 \times 10^5$ | 2373 | **1002** | 2064 |
| $10^6$ | 5347 | **2358** | 4807 |

Table 1: Averaged run time (in second) on random 3-SAT instances. Each solver is performed on each instance within a cutoff time of 10000 seconds so that each run can find a satisfying assignment within the cutoff time.

*Results*

The experimental results are reported in Table 1, which clearly show that the averaged run time of our implementation of WalkSAT is consistently less than half that of WalkSAT_v50. Also, our implementation of WalkSAT performs much faster than probSAT.

## 6. Conclusions

We proposed a new implementation for WalkSAT, which combines a new data structure with the non-caching implementation framework. This implementation enables us to compute the break values of variables very efficiently, and thus improve the performance of WalkSAT. The experiments on random 3-SAT with up to 1 million variables show that our implementation is more than twice faster than existing fastest implementation of WalkSAT, and also significantly outperforms the state-of-the-art SLS solver namely probSAT.

The implementation method in this work can be easily applied to improve other local search algorithms for SAT of focused random walk style.

## References


[1] B. Selman, H. A. Kautz, B. Cohen, Noise strategies for improving local search, in: Proc. of AAAI-94, 1994, pp. 337–343.

[2] L. Kroc, A. Sabharwal, B. Selman, An empirical study of optimal noise and runtime distributions in local search, in: Proc. of SAT-10, 2010, pp. 346–351.

[3] A. Balint, U. Schöning, Choosing probability distributions for stochastic local search and the role of make versus break, in: Proc. of SAT-12, 2012, pp. 16–29.

[4] E. Aurell, U. Gordon, S. Kirkpatrick, Comparing beliefs, surveys, and random walks, in: Proc. of NIPS-04, 2004, pp. 49–56.

[5] S. Seitz, M. Alava, P. Orponen, Focused local search for random 3-satisfiability, J. Stat. Mech. (2005) P06006.

[6] D. Achlioptas, Random satisfiability, in: Handbook of Satisfiability, IOS Press, 2009, pp. 245–270.

[7] H. H. Hoos, T. Stützle, Stochastic Local Search: Foundations & Applications, Elsevier / Morgan Kaufmann, 2004.

[8] H. A. Kautz, A. Sabharwal, B. Selman, Incomplete algorithms, in: Handbook of Satisfiability, IOS Press, 2009, pp. 185–203.

[9] S. Kirkpatrick, B. Selman, Critical behavior in the satisfiability of random boolean formulae, Science 264 (1994) 1297–1301.

[10] A. Balint, D. Gall, G. Kapler, R. Retz, Experiment design and administration for computer clusters for SAT-solvers (edacc), JSAT 7 (2-3) (2010) 77–82.