Efficient Implementations of SAT Local Search

Alex Fukunaga

Computer Science Department, University of California, Los Angeles, CA 90024
fukunaga@cs.ucla.edu

Abstract. Although most of the focus in SAT local search has been on search behavior (deciding which variable to flip next), the overall efficiency of an algorithm depends greatly on the efficiency of executing each variable flip and variable selection. This paper surveys, evaluates, and extends techniques and data structures that have been used in efficient implementations of SAT local search solvers (including GSAT and Walksat variants).

1 Introduction

Local search algorithms for SAT have been studied extensively since the introduction of GSAT [9]. Significant progress in the development of efficient local search algorithms has been made by algorithms such as Walksat [8], Novelty [5], Novelty+ [1], ESG [6], and SAPS [2]. Previously published research in SAT local search has concentrated mainly on the search behavior of the algorithms, i.e., the decision process by which the variable to flip is selected. The precise combination of decision criteria used in the variable selection step (e.g., *breakcount*, defined below), as well as mechanisms such as randomization and clause weighting all impact the trajectory of the algorithm through the space of variable assignments, determining the number of flips required to solve problem instances.

However, the runtime of an algorithm is determined not only by its search behavior, but also the complexity of each iteration of the algorithm. Data structures that enable efficient implementation of variable flips and variable selection have a significant impact on the overall performance of an algorithm, and this is one of the factors for the success of Walksat-family algorithms. This paper surveys and evaluates data structures and algorithms used to support efficient variable flips and selection. We also propose and analyze some extensions to the standard data structures. [1]

2 Analyzing a SAT Local Search Step

SAT local search algorithms generally work as follows: An initial truth assignment is generated (usually randomly). Then, at each iteration, a variable is selected and flipped until either a satisfying solution is found, or some termination condition applies.

Given a candidate variable assignment T for a CNF formula F, let B_0 be the total number of clauses that are currently unsatisfied in F. Let T' be the state of F if variable V is flipped. The *breakcount* of V is the number of clauses which are currently satisfied in T, but will become unsatisfied in T' if V is flipped. The *makecount* of V is the number of clauses which are currently unsatisfied in T, but will become unsatisfied in T' if V is flipped. Let B_1 be the total number of clauses which would be unsatisfied in T'. The *diffscore* of V is $B_1 - B_0$.

We say that a *literal is true* if the current value of the variable is the same as its phase. E.g., if $v_1 = false$, then the negative literal $\overline{v_1}$ is true, while the positive literal v_1 is not true.

Using the above terms: At each iteration, GSAT [9] selects a variable with highest diffscore at each iteration. Walksat [8] first picks random broken clause BC from F. If any variable in BC has a breakcount of 0, then randomly select one of these to flip. Otherwise, with probability p, select a random variable from BC to flip, and with probability $(1-p)$, select the variable in BC with minimal breakcount (breaking ties randomly).

We now consider the complexity of the local search. At each iteration, checking whether the formula is satisfied can be done in constant time by querying the size of an incrementally maintained

[1] Recently, dynamic local search algorithm such as SAPS have been shown to be very successful. In dynamic local search, efficient handling of clause weight updates has a major impact on runtime, but clause weighting is beyond the scope of this paper.

set of unsatisfied clauses. Let the broken (unsatisfied) clause set be implemented as an array of indices or pointers to the broken clauses. Inserting and removing clauses from the broken clause set can be implemented in $O(1)$ time if we also keep track of each clause's position in the broken-clause array. In addition to keeping track of the number of broken clauses, this broken clause array is also used in algorithms such as Walksat to identify a random broken clause in constant time. Therefore, the vast majority of the runtime of SAT local search is spent in the selection of a variable to flip, and in computing the consequences of flipping the variable.

3 Efficient Variable Flips

There is a trade-off between the complexity of the variable flipping and variable selection operations. All current variable selection heuristics involve comparing the breakcount, makecount, and/or diffscore of some subset of variables, or some other function related to these fundamental scores (e.g., such as a weighted diffscore). For example, GSAT requires the identification of the variable in the formula with highest diffscore, while Walksat requires the identification of the variable with lowest breakcount in a single broken clause.

```
def var_flip (v): # flip a variable v
   foreach c in v.linked_clauses_with_positive_lit: clause_update (c, v, +v)
   foreach c in v.linked_clauses_with_negative_lit: clause_update (c, v, -v)

def clause_update (c, v, lit):# compute effect of flipping variable v in clause c
   if (is_literal_true(lit)): #changing a lit from true->false
      if (c.num_true_literals == 1): # case 1->0
         broken_clause_set.add(c)
         foreach var in c: var.makecount += 1
         v.breakcount -= 1
      else if (c.num_true_literals == 2) # case 2->1
         foreach var in c except v:
            if is_literal_true_in_c(var):
               var.breakcount += 1
               break
      c.num_true_literals -= 1
    else: #changing lit from false->true
       if (c.num_true_literals == 0) # case 0->1
          broken_clause_set.remove(c)
          foreach var in c: var.makecount -= 1
          v.breakcount += 1
       else if (c.num_true_literals == 1): # case 1->2
          foreach var in c except v:
             if is_literal_true_in_c(var): var.breakcount -= 1
    c.num_true_literals += 1
```

Fig. 1. Walksat43 Variable Flip procedure

First, consider a *nonincremental* variable flip implementation: At each flip, we change the value of the variable and update the number of satisfied literals for each clause. This takes $O(C/V)$ time per flip. Also, during variable selection, we need to to compute the relevant make/breakcounts for each variable being considered from scratch. Assuming that the number of satisfied literals in a clause can be queried in constant time, then both breakcount and makecount computation take time linear in the number of clauses linked to the variable. For example, consider Walksat on random k-SAT instances. The number of clauses per variable is set to a constant value, C/V (e.g., C/V=4.3 in the phase transition region for the standard 3-SAT benchmarks). Since Walksat selects a variable out of a single broken clause with k variables, each variable selection costs $O(kC/V)$ time. The fastest current implementations of the Walksat-variants (Walksat, Novelty, etc), which are in UBC-SAT-0.9.7[10] uses this nonincremental approach.

An alternate approach incrementally maintains the breakcounts and/or makecounts of all variables, so that each query for the makecount or breakcount of a variable by the variable selection

heuristic is a constant time lookup. The most widely distributed source code for Walksat [4], uses this approach. In addition, the current implementation of SAPS [10] uses this approach.

Fig 1 presents pseudocode for an efficient function for computing the consequences of flipping a variable. based on Walksat version 43 [4]. We call this the Walksat43 clause update procedure.

For each variable, there is an associated vector containing the clauses which contain its positive literal (v.linked_clauses_with_positive_lit), as well as the clauses which contain its negative literal.[2]

To analyze the complexity of `clause_update`, we consider six distinct cases. When flipping v makes lit a satisfied literal, then c has 1, 2, or more than 2 satisfied literals after the flip. We denote these cases by $0 \rightarrow 1$, $1 \rightarrow 2$, $2+$, respectively. Likewise, when flipping v makes lit an unsatisfied literal, then c has 0, 1, or more than 1 satisfied literals after the flip. We denote these cases by $1 \rightarrow 0$, $2 \rightarrow 1$, and $2-$, respectively. The first row in Table 1 shows the complexities of each case for this `Walksat43` algorithm.

	$1 \rightarrow 0$	$2 \rightarrow 1$	$2-$	$0 \rightarrow 1$	$1 \rightarrow 2$	$2+$
Walksat43	O(len)	O(len)	O(1)	O(len)	O(len)	O(1)
watch1	O(len)	O(len)	O(1)	O(len)	O(1)	O(1)
watch2	O(len)	O(1)	O(len)	O(len)	O(1)	O(1)
Walksat43-break-only	O(1)	O(len)	O(1)	O(1)	O(len)	O(1)
watch2-break-only	O(1)	O(1)	O(len)	O(1)	O(1)	O(1)

Table 1. Complexity of clause update cases. *O(len)* = linear in the length of the clause

It is relatively straightforward to improve upon the Walksat43 clause update scheme. When a clause c only has one satisfied literal, the variable for that literal can be stored and associated with the clause. We call this the *watch1* strategy, because we "watch" 1 satisfied literal per clause. This reduces the complexity of the case $1 \rightarrow 2$ from O(len) to O(1), and offers a straightforward improvement over the Walksat43 scheme. The SAPS implementation in UBC-SAT 0.9.7 uses this strategy.

It turns out that it is possible to do even better than *watch1* by exploiting the runtime behavior of local search algorithms. We performed an experiment to observe the frequencies with which the 6 diffscore transition cases were occurring. Each time `clause_update` was called, we incremented one of 6 counters corresponding to each of the transition cases. We used 5 test instances: `bw_large.c` is an AI planning problem instance, `uf250-98` is a random 3-SAT instance with 250 variables, `flat-200-10` is a graph coloring instance, and `e0ddr2-10-by-5-1` and `dlx2_cc_a_bug17` are processor verification instances. All instances are from SATLIB (www.satlib.org), except the `dlx2_cc_a_bug17`, which is from [11]. For each instance, we ran Walksat up to 300,000 flips (or until the instance was solved). At the end of each run, we computed the frequencies of each type of diffscore transition. The results in Table 2 are the means over 10 runs on each instance. Interestingly, the $2 \rightarrow 1$ and $1 \rightarrow 2$ transitions are by far the most frequent transitions.[3]

instance	# vars	# clauses	$1 \rightarrow 0$	$2 \rightarrow 1$	$2-$	$0 \rightarrow 1$	$1 \rightarrow 2$	$2+$
bw_large.c	3016	50457	3.75%	40.66%	5.38%	3.88%	40.93%	5.39%
uf-250-02	250	1065	9.21%	28.55%	12.21%	9.25%	28.56%	12.22%
dlx2_cc_a_bug17	4847	39184	1.25%	28.56%	20.08%	1.40%	28.77%	19.94 %
e0ddr2-10-by-5-1	19500	103887	7.53%	41.05%	1.41%	7.68%	41.00 %	1.32 %
flat200-10	600	2237	13.04%	36.94 %	0.00 %	13.07%	36.96 %	0.00 %

Table 2. Diffscore transition case frequencies in Walksat

We can exploit the heavily skewed distribution of transition frequencies by trying to speed up the most frequent cases. Figure 2 shows *watch2*, a new algorithm for `clause_update`. It turns

[2] This is more efficient than merely keeping a single vector of all clauses containing a literal of v, since we can then pass *lit* into `clause_update`, instead of computing it (which would require either 1) iterating through all the literals in c to find out which of them was a literal of v, or 2) keeping a table which took c and v as indices and returns whether c contains the positive or negative literal of v.

[3] Fig. 2 shows the transition frequencies for all clauses. If binary clauses were excluded, the results remain very similar, except for the case of `flat-200-10`, where the relative frequency of the $2 \rightarrow 1$ transition drops to almost zero when binary clauses are excluded. However, the $1 \rightarrow 2$ frequency remains 0.

out that by keeping track of a *second* satisfied literal for each clause, it is possible to reduce the complexity of the $2 \rightarrow 1$ case from O(len) to O(1). The tradeoff is that the $2-$ case becomes O(len). The complexities of the other cases remain unchanged. This is a worthwhile tradeoff, since we have shown empirically that the $2 \rightarrow 1$ case is much more common than the $2-$ case. In Figure 2, `clause.true_lit1` and `clause.true_lit2` are assigned as follows: If c has no satisfied literals, then both are NULL. If c has at least one satisfied literal, then `clause.true_lit1` is assigned. If c has at least 2 satisfied literals, then both `clause.true_lit1` and `clause.true_lit2` are assigned. Table 1 shows the complexity for each diffscore transition.

```
def clause_update_watch2(c,v,lit):
   if (is_literal_true(lit):
      if (c.num_true_literals == 1): # case 1->0. clause becomes broken
         broken_clauses.add(c)
         foreach var in c: var.makecount += 1
         v.breakcount -= 1
      else if (c.num_true_literals == 2): # case 2->1
         if (c.true_lit1 == lit): c.true_lit1 = c.true_lit2
         var = var_of_lit(c.true_lit1)
         var.breakcount += 1
      else:           # case 2-
         # if lit is being watched, then must find a replacement
         if (lit == c.true_lit1): # find lit other than true_lit1, true_lit2
            c.true_lit1 = c.find_new_true_literal()
         else if (lit == c.true_lit2):
            c.true_lit2 = c.find_new_true_literal()
      c.num_true_literals -= 1
   else:
      if (c.num_true_literals == 0): # case 0->1
         broken_clauses.remove(c)
         foreach var in c: var.makecount -= 1
         v.breakcount += 1
         c.true_lit1 = lit
      else if (c.num_true_literals == 1): # case 1->2
         var = var_of_literal(c.true_lit1)
         var.breakcount -= 1
 c.true_lit2 = lit
      c.num_true_literals += 1
```

Fig. 2. Watch2 Clause update procedure (replaces `clause_update` in Fig 1

Not all SAT local search algorithms require both makecount and breakcount. Walksat only requires breakcount. Thus, `clause_update` can be faster without makecount updates. In Table 1, *Walksat43-break-only* and *watch2-break-only* show the complexities of the *Walksat43* and *watch2* schemes when makecount computation is eliminated.

We compared the performance of the *Walksat43*, *watch1*, and *watch2* algorithms. For each clause update scheme, Walksat with noise 0.5 was executed on each benchmark for 100 runs, where each run was up to 300,000 flips. We reset the random seed for each algorithm, so that the search behavior was identical (i.e., the timings reflect only the performance differences resulting from the variable flip efficiency). **Total runtimes** are shown in Table 3.[4]

Note that we purposefully used the Walksat algorithm (which doesn't require makecount) instead of one of the Novelty-family algorithms (which require both makecount and breakcount to compute diffscore), because we also wanted to show the impact of makecount computation. *watch2-break-only* shows the results with makecount computation removed from *watch2*. Again, this variant performs the exact same search steps as the the versions with makecount updates so the runtime differences are solely due to the differences in efficiency in the clause update procedure.

[4] The experiments were run on a 1.7GHz Athlon. Note that our code was implemented in Common Lisp using the CMUCL compiler. When the code is optimized with type declarations, CMUCL generates code that is usually within a factor of 2-4 of C code.

Random-Walk is a straw man algorithm that, at each iteration, picks a broken clause and flips a random variable from the clause. This requires neither breakcount nor makecount updates, and has constant time complexity for all of the transitions in Table 1. Although Walksat and Random Walk have very different search behaviors, the comparison is useful because it *Random-Walk* indicates an approximate lower bound on the runtime that can be achieved in our implementation framework if breakcount and makecount updates cost nothing. In Figure 3, we only show the results for the larger instances, for which Walksat does not find solutions within the 300K flip limit (that way, we are at least comparing the runtimes for the same number of flips as Walksat, even though they are for different search trajectories).

As expected, the *watch1* and *watch2* clause update procedures performed significantly better (up to 50% faster) compared to the baseline *Walksat43* clause update procedure. Compared to the approximate lower bound indicated by *Random-Walk*, *watch2* procedure is within a factor of 2.

We have recently implemented the *watch2* strategy in the Walksat v43 code and obtained similar results. The most recent version, Walksat v45, incorporates this patch. Similarly, we have also implemented *watch2* in the UBCSAT0.9.7 framework, and have obtained a 5-20% speedup on the SAPS algorithm (which used *watch1* in version 0.9.7).

instance	Walksat43	Watch-1	Watch-2	Watch-2 Break-only	Random Walk
bw_large.a	7.62	6.85	6.77	4.98	n/a
bw_large.c	638.67	613.37	458.03	392.14	316.57
uf-250-02	7.93	6.91	7.14	5.01	n/a
dlx2_cc_a_bug17	278.71	236.2	153.02	151.30	n/a
e0ddr2-10-by-5-1	1458.53	1279.45	1038.28	775.55	523.37
flat200-10	52.58	46.31	49.45	33.34	19.4

Table 3. Comparison of variable flip procedure implementations (**total runtimes** for 100 runs)

4 Efficient Variable Selection

The previous section focused on data structures allowing efficient queries for the breakcount/makecount/diffscore of any particular variable. Now, we consider the orthogonal problem of selecting a variable from a set of variables given a particular criterion. Broadly speaking, there are two classes of variable selection strategies:

The first class of strategies, exemplified by Walksat, selects a variable from a Small subset of variables in the formula (in the case of Walksat, those belonging to a single broken clause). Selecting a variable requires iterating through each variable in the clause, identifying the one that has the lowest breakcount. The cost of variable selection scales linearly with the length of the clauses in the formula. In some benchmarks (e.g., random 3-SAT) this is a small constant, and therefore the selection complexity of Walksat-family algorithms scales very well on these benchmarks. In formula derived from real-world problems, the size of the largest clauses tends to scale with the size of the problem. However, there are many classes of problems where the *average* size of the clauses remains small (e.g., the planning problem instances have many binary clauses). When average clause length is relatively short, then iterating through each of the variables and querying its make/break/diffscore values is not very expensive. In fact, profiling on our benchmark formulas confirms that the majority of the runtime in Walksat is being spent on clause updating.

Variable selection complexity becomes a significant issue in the second class of algorithms, exemplified by GSAT and SAPS. These select a variable from a much bigger set of variables (e.g., the set of all variables in the formula, or variables that are in *any* broken clause). The original GSAT algorithm requires identifying the variable with maximum diffscore. The simplest approach is to scan the set of all variables, for the variable with best diffscore. However, this takes $O(V)$ time, and as the number of variables grows, this scanning operation becomes an increasingly heavy price to pay per flip, and profiling shows that for large V, this can dominate the runtime of the algorithm. The UBC-SAT0.9.7 implementation of SAPS, a state of the art algorithm, maintains the set of all variables that have positive makecount, and scans that set to find the best variable according to its weighted clause metric.

Another approach (used in [7]) is to incrementally maintain *MAX-DIFF-VARS*, the set of variables with maximum diffscore. This scheme, which depends upon incremental maintenance of make/breakcounts, allows constant time retrieval of the max diffscore variable. Incremental update operations are fast as long as the max diffscore does not decrease, but when max diffscore decreases, a linear scan through the variables is required. For algorithms that depend on clause-weighting, this technique is not applicable because of the granularity of the score.

Another approach (originally implemented in GSAT version 25[7]) dynamically partitions the variables into three subsets, where the three subsets contain the variables with positive, zero, and negative diffscore, respectively. Whenever the diffscore of a a variable changes, it is moved between the three buckets (in constant time). All variables in the same bucket are treated as being equivalent for the purposes of variable selection. If there are any variables in the positive bucket, then one is chosen randomly, else if there are any variables in the zero bucket, then one is chosen randomly, otherwise, a negative diffscore variable is returned. Iwama et al [3] noted that it is not necessary to actually maintain a negative bucket, since if a variable is not in the positive or zero buckets, then its presence in the negative bucket is automatically implied. Using this bucket data structure reduces variable selection from linear to constant time. However, by treating all variables in the same bucket as equivalent, the three (or two) bucket implementation changes the search behavior and makes the algorithm less greedy (and is no longer equivalent to GSAT).

5 Conclusions

This paper reviewed and evaluated techniques and data structures used in efficient SAT local search implementations. We also proposed a new incremental variable flipping, *watch2*, which provided an improvement over state of the art implementations. While search algorithm behavior has been the focus of most past SAT local search research, implementation techniques such as those covered here can significantly impact the runtimes of the algorithms, and new algorithms should be considered with this in mind. Finally, implementation techniques from older search procedures such as GSAT may prove to be valuable in new contexts. Some recent state of the art algorithms such as SAPS are similar to GSAT in that variables must be selected from a larger pool of variables than the Walksat-architecture algorithms. Thus, techniques used in efficient GSAT implementations (c.f., Sec. 4) may be useful for these newer algorithms.

References

1. H. Hoos and T. Stutzle. Local search algorithms for sat: An empirical evaluation. *Journal of Automated Reasoning*, 24:421–481, 2000.
2. F. Hutter, D. Tompkins, and H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In *Proceedings of CP-02*, pages 233–248. Springer-Verlag, 2002.
3. K. Iwama, D. Kawai, S. Miyazaki, Y. Okabe, and J. Umemoto. Parallelizing local search for cnf satisfiability using vectorization and pvm. *ACM Journal of Experimental Algorithms*, 7(2), 2002.
4. H. Kautz and B. Selman. Walksat version 43. http://www.cs.washington.edu/homes/kautz/walksat/walksat.
5. D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proc. AAAI*, pages 459–465, 1997.
6. D. Schuurmans, F. Southey, and R.C. Holte. The exponentiated subgradient algoithm for heuristic boolean programming. In *IJCAI*, pages 334–341, 2001.
7. B. Selman and H. Kautz. Gsat v35 user's guide. http://www.cs.washington.edu/homes/kautz/walksat/gsat-tar-Z.uu.
8. B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. AAAI*, 1994.
9. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proc. AAAI*, pages 440–446, 1992.
10. D. Tompkins. Ubcsat. http://www.cs.ubc.ca/ davet/ubcsat.
11. M.N. Velev. Superscalar suite 1.0a. Available from: http://www.ece.cmu.edu/ mvelev.