Learning Short-Term Weights for GSAT

Jeremy Frank *
frank@ptolemy.arc.nasa.gov
Caelum Research Corp.
NASA Ames Research Center Mail Stop 269-2
Moffett Field, CA. 94035-1000

Abstract

We investigate an improvement to GSAT which associates a weight with each clause. We change the objective function so that GSAT moves to assignments maximizing the weight of satisfied clauses, and each clause's weight is changed when GSAT moves to an assignment in which this clause is unsatisfied. We present results showing that this version of GSAT has good performance when clause weights are reduced geometrically throughout the course of a single try. We conclude that clause weights are best interpreted as short-term, context sensitive indicators of how hard different clauses are to satisfy.

1 Introduction

Local search procedures are an alternative to complete search algorithms for solving combinatorially expensive search problems. GSAT is a local search algorithm which can often find solutions to satisfiable SAT problems in Conjunctive Normal Form (CNF) quickly [SLM92]. GSAT operates by changing a complete assignment of variables into one in which the maximum possible number of clauses are satisfied by flipping a single variable. GSAT can quickly solve satisfiable 3-CNF problem instances of up to 2000 variables which are known to have solutions, while the best known complete search procedures typically are able to solve problems with 500 variables or less.

A feature of GSAT is that often the best move it can make leaves the number of unsatisfied clauses the same. These moves are known as "sideways" moves, and GSAT typically searches regions of sideways moves for some time before finding a move which reduces the number of unsatisfied clauses. This means GSAT must randomly search these "plateaus" until it finds a way off. Much research into GSAT has focused on reducing the impact of plateau search, typically by encouraging exploration of parts of the space that GSAT has not explored yet.

Several modifications of GSAT have been developed to reduce the impact of searching the plateaus. Tabu search [Glo89], [HdW87] is one such scheme; when a variable is flipped, it is placed on a *tabu list* and can't be flipped again for a constant amount of time. This is a primitive learning scheme which remembers actions taken in the recent past and does not reverse them. Tabu search has been augmented with a variety of schemes to allow reversing flips under certain circumstances.

Selman and Kautz [SK93] added random walk to GSAT, enabling GSAT to make backwards moves in an attempt to escape plateaus. Gent and Walsh [GW95] developed a wide variety of modifications to GSAT and tested many of these with random walk. Spears [Spe95] discusses a modification of GSAT that employs simulated annealing to allow escape from local minima. In this modification the change in the number of satisfied clauses and a "temperature" parameter are used to compute the probability that some variable flip is accepted; as more flips are done the probability that backwards moves are selected decreases.

A modification of local search that has received recent attention is the "clause weights" scheme proposed in [SK93] and in [Mor93]. In this version, each clause has an associated weight, and these weights are modified during search in order to better inform the variable selection heuristic. The criterion for variable selection becomes "maximize the weight of satisfied clauses". This variant of GSAT has been investigated by Cha and Iwama as a stand-alone modification[CI95] and in conjunction with a method to add additional clauses to problem instances[CI96]. Davenport et. al. use a similar scheme in GENET applied to a connectionist architecture [DTWZ94]. Frank [Fra96] analyzed several variants

*This research was supported by NSF CCR-94-0365. I would like to thank Peter Cheeseman, John Stutz and John Allen of NASA Ames Research Center for their input during this research, and also Ian Gent of the University of Strathclyde, Toby Walsh of IRST and the IJCAI reviewers for their comments.

based on Selman and Kautz' approach.

We present a modification of weighted GSAT and describe different algorithms for modifying the clause weights. We interpret the process of modifying weights as *learning* how difficult a clause is to satisfy, and by extension which variables to focus on at each flip. We show that the best learning algorithm retains a context-sensitive weight for each clause rather than allowing clause weights to grow unbounded. Our interpretation is that as the assignments GSAT examine change, the weight should represent the difficulty of satisfying the clauses in the context of recent assignments. We also demonstrate a tradeoff between the rate of learning and the amount of contextual information preserved.

In §2 we discuss variants of weighted GSAT. In §3 we discuss the experiments we ran to gather empirical data on the performance of our variants, and our analysis is presented in §4. In §5 we discuss conclusions and future work.

2 GSAT and Variants

We give the familiar algorithm outline for GSAT in figure 1. GSAT is a local search algorithm which typically employs greedy hill-climbing [SLM92] designed to solve CNF formulae. GSAT begins with a randomly generated initial truth assignment, then hill-climbs by reversing or "flipping" the assignment of the variable which increases the number of satisfied clauses the most. If our CNF Σ has C clauses, then $eval = \sum_{i=1}^{C} Sat_i(A)$ where $Sat_i(A) = 0$ if clause i is unsatisfied in assignment A and 1 if it is satisfied. If PossFlips contains more than one variable, the algorithm must now pick one variable to flip. Different GSAT variants control how many and which variables are examined to determine the best flip and tie-breaking schemes to pick among equally good flips.

Selman and Kautz [SK93] describe a modification in which GSAT associates a weight with each clause. The weights of all clauses that remains unsatisfied at the end of a try are incremented. A clause that has remained unsatisfied through many tries will have a higher weight than a clause which has been satisfied in most of those tries. Each clause's weight is initialized to 1, so the first try is controlled as it would be in GSAT. GSAT then picks the flip that satisfies clauses with the highest total weight. So if W_i denotes the weights of the clauses then our function *eval* can be written $eval = \sum_{i=1}^{C} W_i * Sat_i(A)$.

We saw several modifications which we felt might improve the algorithm's performance and illuminate the effect that weights have on GSAT's performance, leading to even better improvements in the future. In particular, we feel that clause weights can be used to *learn* which parts of the problem instance are hard to solve,

```
procedure GSAT(Σ,MaxFlips, MaxTries)
  for i=1 to MaxTries
    A = gen_assignment
    for j = 1 to MaxFlips
      if solved_problem(A)
        return A
      else PossFlips = select(Σ,A)
        V=pick(PossFlips)
        A=A with V's value flipped
      end else
    end for
  end for
end

procedure select(Σ,A)
PossFlips=ε
Best = -∞
  for i=1 to Number of Vars
    A=A with V_i's value flipped
    if eval(A, Σ) =Best
      PossFlips=PossFlips ∪ V
    else if eval(A, Σ)>Best
      PossFlips=V
      Best = eval(A, Σ)
    A=A with V_i's value flipped
  end for
end
```

Figure 1: GSAT Algorithm Sketch.

which will improve the performance of weighted GSAT. We briefly describe these modifications below.

2.1 When to Change the Weights?

Selman and Kautz originally suggested altering the weights after each try. This has the effect of allowing GSAT to drive the number of satisfied clauses to a very low number relative to the initial number; as a result, few weights change after each try and feedback must wait until the end of a try. We decided to update the clause weights after each *flip* instead of after each try. Unsatisfied clauses provide feedback to the variable selection heuristic every flip. This has the effect of emphasizing variables in unsatisfied clauses almost immediately, without waiting for the try to complete. If we view the weight modification process as learning, we see that this version starts learning immediately rather than waiting until each try is complete, thereby acquiring information about clauses more rapidly. After each flip we update the clauses in the following way: if $W_{i,t}$ denotes the weight of clause i at time t then $W_{i,t} = W_{i,t-1} + \delta$ if clause i not satisfied and $W_{i,t} = W_{i,t-1}$ otherwise. This scheme resembles that used by Morris [Mor93], who changes weights when the best move leaves the value of the objective function unchanged. If a try is unsuccessful then we reset the weights to 1 before starting the new try. We shall refer to this algorithm as WGSAT, not to be confused with WSAT, the random walk version of GSAT.

Different learning rates may make a difference in

WGSAT's performance [SK93]. Suppose a clauses' weight increases to x after the first flip. This clause now "outweighs" x clauses of weight 1. The heuristic prefers to satisfy clauses of higher weights, and increasing the increment size can result in different moves, and therefore varying performance of the algorithm. In a learning context, a larger increment implies "fast" learning or "focusing", in that larger weights mean that the algorithm rewards satisfaction of those highly weighted clauses more quickly than if the clauses are incremented at a more moderate rate. If the rate of learning is too fast then WGSAT may "thrash" by making moves which almost always unsatisfy more clauses than necessary. However, if the rate of learning is too slow then WGSAT will require more flips to solve problem instances.

2.2 Short Term Weights

One observation we made about clause weights is that their *relative magnitudes* control which state is best to move to. If a single clause has a high weight, it may be able to sway the heuristic to cause several clauses of lower weight to become unsatisfied, but can't sway a clause of higher weight than its own. The original WGSAT algorithm never decreases the weights once they are incremented, so after a time it becomes very difficult to change the relative magnitudes of the weights. In WGSAT clauses accumulate weight in the context of assignments which are constantly changing. It may be that weights acquired during one part of search will hinder later parts of the search.

To combat this problem we implemented two schemes to reduce weights throughout local search. The first reduces clause weights if the clause was satisfied in an assignment. For example, if a clause was satisfied in the current assignment we could reduce its weight by 1, while if it was unsatisfied then we could increase its weight by 5. We envision a large increment when a clause is unsatisfied so that the clauses weight will endure for a time, even if it is satisfied in the next several assignments. This gives the clause weight scheme a feel similar to that of tabu search [Glo89], [HdW87], where variables are not allowed to be flipped for some fixed duration. This scheme is more general than tabu search, because it provides a continuously weakening incentive not to flip the variable instead of disallowing the flip. We can analogize decrementing weights to "refocusing" on other clauses which have not been satisfied by changing the relative weight of satisfied clauses faster. The weight update scheme has the form $W_{i,t} = W_{i,t-1} - \delta_s$ if clause i is satisfied and $W_{i,t} = W_{i,t-1} + \delta_u$ if clause i not satisfied. We refer to δ_u as the refocusing rate to distinguish it from δ_s the learning rate for this version. In our example above, $\delta_u = 5$ and $\delta_s = 1$. Unfortunately, this scheme was not successful and due to space considerations we do not discuss it further here.

The second scheme only stores the increments which took place *recently* instead of keeping all clause weight changes. In this scheme clause weights reflect how difficult this clause was to satisfy only in the context of recent assignments. The general form of this scheme is $W_{i,t} = p * W_{i,t-1} + \delta$, where we refer to p as the decay rate. We restrict $p < 1$ and the learning rate $\delta \geq 0$. With this scheme the maximum weight is $\frac{\delta}{1-p}$, which has the side effect of preventing unbounded growth of weights due to monotone increasing as in the previous schemes. The choice of decay rate will dictate how extensive the context is, and at what rate older weight information is "forgotten".

3 Experiments

We tested our algorithms on problems found near the phase transition for 3-CNF [CKT91]; unless otherwise stated, the algorithms were tested on problems for which the number of clauses C and the number of variables N obeyed the constraint $\frac{C}{N} = 4.3$. We created the problem instances randomly; that is, we generated each clause of each problem by selecting 3 of N literals without replacement and negating each literal with probability $\frac{1}{2}$. We generated test suites of problems which were proven to have solutions using a variant of the Davis-Putnam procedure.

We employed two different metrics when analyzing the performance of the algorithm variants. The first metric is the proportion of problem instances from the test suite which the variant solved in a single try. This metric is appropriate for the situation where a practitioner is solving problems from an unknown distribution. The second metric is the minimum mean time to solve all problem instances in the test suite. Gent and Walsh analyze versions of GSAT by finding the value of MaxFlips minimizing the mean time to solve a test suite of known solvable problems [GW95]. This metric is appropriate when a practitioner knows something about the distribution of problems and has tuned the value of MaxFlips to minimize the time required to solve problem instances from the distribution. HSAT [GW93], which selects the variable flipped longest ago if there are several variables resulting in the best change in the number of satisfied clauses, has among the best performance with MaxFlips tuned. We therefore use HSAT as a benchmark algorithm and test our variants against the performance of HSAT.

3.1 Effectiveness of Decaying Weights

We implemented WGSAT with decay in order to test its performance. We found that the learning rate makes little difference in the single try performance of WGSAT without decay, so we only present results on combining different learning rates with decay. As we stated before, if weights are updated according to the formula $W_{i,t} = p * W_{i,t-1} + \delta$ then the maximum weight is $\frac{\delta}{1-p}$ and the maximum weight after F flips is $\frac{\delta(1-p^F)}{1-p}$. We tested WGSAT with decay rates $p = 0.95, 0.99, 0.999$ and learning rates of $\delta = 1, 8, 16, 24$. The plots in figures 2 to 4 present the data for fixed decay rates; for comparison we also include the plot of WGSAT without decay in each figure. We see in figure 2 that when p is close to 1, in this case 0.999, that even these small differences disappear and the number of solved problems rises for all values of MaxFlips. When the decay rate grows further

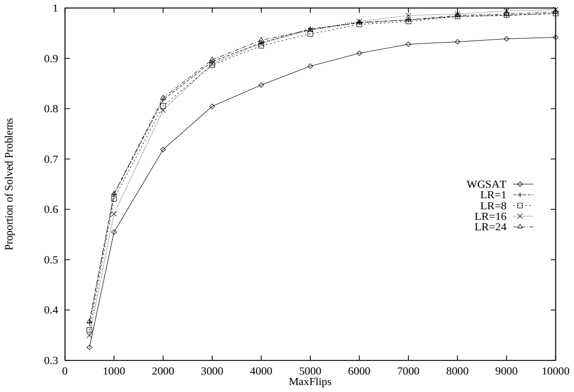

Figure 2: Satisfiability Performance of WGSAT With Varying Learning Rates and Decay Rate of 0.999.

from 1, as shown in figures 3 and 4, we notice that for some pairs of learning rates and decay rates the number of solved problems decreases. For instance in figure 4 we see that the performance of the variant with $\delta = 1$ solves more problems for small numbers of MaxFlips than the other variants. We believe that the other variants emphasize clauses too quickly, resulting in "thrashing" until the weights settle. However, as MaxFlips increases, the

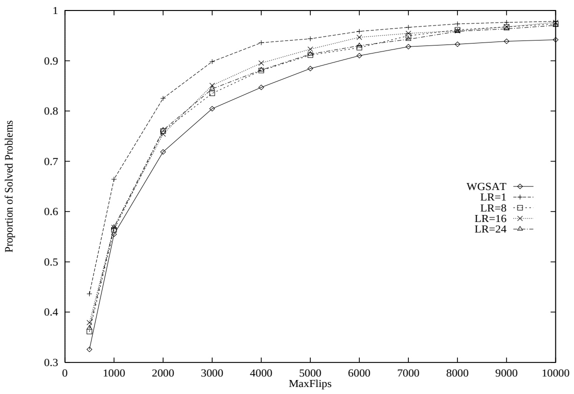

Figure 3: Satisfiability Performance of WGSAT With Varying Learning Rates and Decay Rate of 0.99.

variant with $\delta = 1$ becomes much worse than the rest of the variants. This indicates that when the learning rate is higher, WGSAT can tolerate a faster loss of weights before losing performance. By contrast, in figure 3 we see that the variant with $\delta = 1$ performs *better* than the rest of the variants. Our explanation is that with higher δ and p very close to 1, weights accumulate too fast resulting in over-emphasizing clauses too early in search. We see that the performance of WGSAT with the decay rate very close to 1 is the best version. Figure 5 shows the performance of WGSAT with $\delta = 1$ for different decay rates and for no decay. Notice that when $p = 0.95$ and MaxFlips is low WGSAT outperforms the other variants, but as MaxFlips increases the number of solved problems for $p = 0.95$ is smaller than for p closer to 1. This implies that it is important to discount weights acquired early in the process, but more important to preserve weights acquired later.

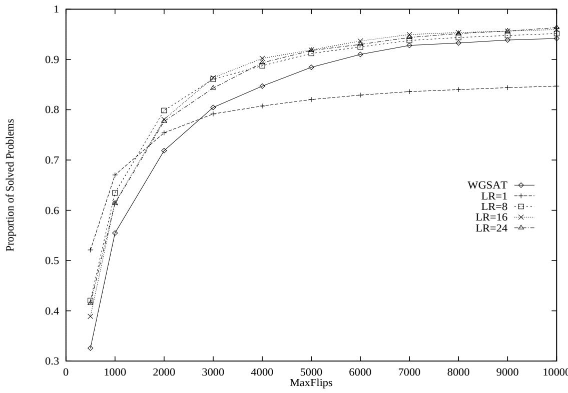

Figure 4: Satisfiability Performance of WGSAT With Varying Learning Rates and Decay Rate of 0.95.

Our analysis indicates that using a decay scheme with WGSAT might improve the optimal performance of WGSAT. Gent and Walsh [GW95] show that GSAT variants require variable amounts of time to solve problems for different values of MaxFlips. Accordingly we tested 3 variants of GSAT to find the value of MaxFlips which minimizes the mean number of flips to solve a test suite of 1000 known solvable problems. We allowed each variant of GSAT enough tries to guarantee that all problems were solved, then computed the mean number of flips required to solve the suite. We tested WGSAT without decay, WGSAT using a decay rate of 0.999, and HSAT. Both variants of WGSAT used a learning rate of 1. We ran our experiments on problems of 100, 125, 150 and 175 variables. The results are presented in figure 6. We see that the performance of WGSAT and WGSAT with decay are roughly comparable until we solved problems of 150 variables. This is somewhat surprising given that for a fixed number of flips WGSAT

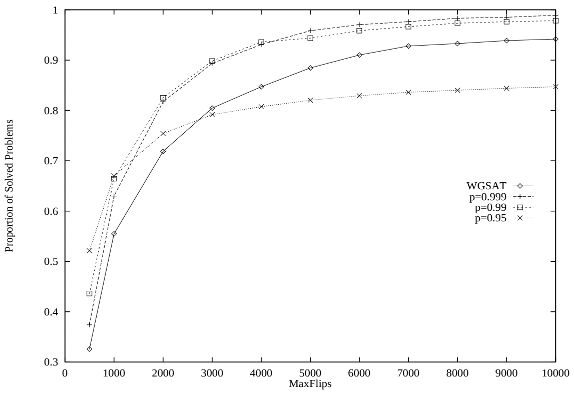

Figure 5: Satisfiability Performance of WGSAT With Varying Decay Rates and Learning Rate of 1.

Vars	Variant	Opt MaxFlips	Mean	SDev
100	WGSAT	1250	1591.95	2346.2
	WGSAT+Decay	1000	1616.84	2322.05
	HSAT	200	2437.27	4269.09
125	WGSAT	1750	3454.23	5159.49
	WGSAT+Decay	1700	3511.1	5721.7
	HSAT	300	5480.54	10220
150	WGSAT	2000	6982.04	12366.2
	WGSAT+Decay	1900	6733.4	11510.6
	HSAT	550	9836.46	17697
175	WGSAT	2800	13169.2	37702.7
	WGSAT+Decay	2600	12081.2	21157.6
	HSAT	700	22283.2	40605.1

Figure 6: Optimal Performance of GSAT Variants.

with decay solved more problems than WGSAT for problems of 100 variables and 125 variables. However, we notice that the best performance of WGSAT with decay occurs at a lower value of MaxFlips than the optimal value of MaxFlips for WGSAT with no decay. For problems of 150 variables WGSAT with decay begins to show a slight improvement in the mean number of flips as well as the variance. The best improvement appears for problems of 175 variables; we see that WGSAT with decay shows a definite improvement over WGSAT. Both versions are considerably better than HSAT for all problem sizes. We note that HSAT is somewhat better than versions of GSAT with Random Walk (results supporting this conclusion come from [GW95]), so improvements upon HSAT are strong results of the effectiveness of our variants.

3.2 Structured Problems

Numerous researchers have also analyzed the performance of GSAT and GSAT variants on structured SAT problem instances from planning and scheduling applications. We analyzed WGSAT and WGSAT with decay on several problem instances taken from standard problem instance sets used by other researchers. Due to the performance differences observed for problems of variable size we tested problems of a variety of different sizes. The instances are: satisfiability encodings of a simple logistics problem with 331 variables and 2246 clauses (Rocket) and a large blocks world problem with 459 variables and 4675 clauses (Large) from the SATPLAN library [KS96], a satisfiability encoding of the 3-disk Tower of Hanoi problem with 158 variables and 992 clauses (Tower), and a circuit diagnosis problem with 331 variables and 712 clauses from the DIMACS archive (SSA). For each problem instance we ran WGSAT with $\delta = 1$, prior=1 and $\alpha = 1$ and WGSAT with decay with $\delta = 1$ and $p = 0.999$. We set MaxFlips at 10000 and ran 100 tries; we present the number of successful tries and the mean and variance in the number of flips required to solve the problem instances for those successful tries.

WGSAT			
Problem	Solved	Average flips	Sdev flips
Tower	28	2900	3881
SSA	41	374	73
Large	76	3587	4324
Rocket	70	2626	3269
WGSAT with Decay			
Problem	Solved	Average flips	Sdev flips
Tower	32	3167	3622
SSA	44	370	68
Large	70	3799	4491
Rocket	63	2764	3551

Figure 7: Performance of WGSAT Variants on Structure SAT Problems.

GSAT and HSAT were unable to solve these problems in 100 tries for many settings of MaxFlips. Figure 7 shows the proportion of tries out of 100 that WGSAT variants solved the problems, the mean number of flips required to solve the problems on successful attempts, and the standard deviation, also on successful attempts. WGSAT variants can solve these problem instances with some regularity, so the modifications clearly have merit beyond applicability to randomly generated problems. WGSAT and WGSAT with decay perform comparably on these problem instances. WGSAT appears to be slightly better on the SAT encoding of the 3-disk Tower of Hanoi problem, while WGSAT with decay appears to be better on the Circuit Diagnosis problem, but the differences are slight.

4 Analysis

The results show that gradually decaying weights can improve the performance of WGSAT compared to not decaying weights for randomly generated problems as the problem size increases. We noticed that when the learning rate was high we can decay rates more rapidly and maintain good performance, but the best version seems to be a decay rate close to 1, and at this value the learning rate appears irrelevant. Our conclusion is that clause weights are best interpreted as a context dependent measurement of the difficulty of satisfying a clause, and as search moves away from the current context the weight should represent the difficulty of satisfying the clauses in the new context. However, since the context changes slowly the weights must also be decayed slowly. The results showing that rapid decay is effective early in the search indicates that WGSAT can suffer from prematurely weighting clauses, but the reduced effectiveness of rapid decay for higher MaxFlips indicates that we must walk a fine line between no decay and too much decay.

5 Conclusions and Future Work

We have presented a modification of an update to GSAT which uses clause weights to modify the GSAT variable selection heuristic continuously throughout local search. We have demonstrated that decaying these weights gradually throughout search results in a more effective version of GSAT than several previous versions. We have also performed experiments which assess the impact of different weight updating schemes and concluded that gradual decay of weights gives the best results. We interpreted these methods as learning the best way to conduct local search by discovering the hardest clauses to satisfy relative to recent assignments.

Currently none of our variants of WGSAT carry weights over to subsequent tries. We would like to combine information from previous tries in new tries for WGSAT. Our limited experiments found that using the weights found on a previous try can lead to occasional spectacular failures resulting in millions of flips required to solve instances. One idea is to "discount" the weights from previous tries, in effect using the previous try as a non-uniform prior weight. An important consideration in new versions of GSAT is the tradeoff between reduced numbers of flips required to solve problems and the CPU time required to run these schemes.

Many of the schemes presented in this paper show improvements over GSAT and HSAT, but require floating-point manipulations where GSAT and HSAT use only integer (or even bit) computations. Cache and memory performance issues can also play a role, since schemes like HSAT and WGSAT require variable amounts of memory. We are conducting investigations into the impact these issues have on the real time performance of GSAT variants.

References

[CI95] B. Cha and K. Iwama. Performance tests of local search algorithms using new types of random cnf formulas. *14th International Joint Conference on Artificial Intelligence*, pages 304–310, 1995.

[CI96] B. Cha and K. Iwama. Adding new clauses for faster local search. *Proceedings of the 13th National Conference on Artificial Intelligence*, 1996.

[CKT91] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the *really* hard problems are. *IJCAI*, pages 163–169, 1991.

[DTWZ94] A. Davenport, E. Tsang, C. J. Wang, and K. Zhu. Genet: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 325–330, 1994.

[Fra96] J. Frank. Weighting for godot: Learning heuristics for gsat. *Proceedings of the 13th National Conference on Artificial Intelligence*, 1996.

[Glo89] F. Glover. Tabu search part i. *ORSA Journal on Computing*, 1989.

[GW93] I. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for sat. *AAAI*, pages 28–33, 1993.

[GW95] I. Gent and T. Walsh. Unsatisfied variables in local search. In J. Hallam, editor, *Hybrid Problems, Hybrid Solutions.* IOS Press, 1995.

[HdW87] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39:345–351, 1987.

[KS96] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. *Proceedings of the 13th National Conference on Artificial Intelligence*, 1996.

[Mor93] P. Morris. The breakout method for escaping from local minima. *AAAI*, 1993.

[SK93] B. Selman and H. Kautz. Domain independent versions of GSAT solving large structured satisfiability problems. *IJCAI*, 1993.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. *AAAI*, pages 440–446, 1992.

[Spe95] W. Spears. Simulated annealing for hard satisfiability problems. In D. Johnson and M. Trick, editors, *2d. DIMACS Implementation Challenge*, DIMACS Series in Theoretical Computer Science, 1995.