Trap Escaping Strategies in Discrete Lagrangian Methods for Solving Hard Satisfiability and Maximum Satisfiability Problems*

Zhe Wu and **Benjamin W. Wah**
Department of Electrical and Computer Engineering
and the Coordinated Science Laboratory
University of Illinois, Urbana-Champaign
Urbana, IL 61801, USA
E-mail: {zhewu, wah}@manip.crhc.uiuc.edu
URL: http://www.manip.crhc.uiuc.edu

Abstract

In this paper, we present efficient trap-escaping strategies in a search based on discrete Lagrange multipliers to solve difficult SAT problems. Although a basic discrete Lagrangian method (DLM) can solve most of the satisfiable DIMACS SAT benchmarks efficiently, a few of the large benchmarks have eluded solutions by any local-search methods today. These difficult benchmarks generally have many traps that attract local-search trajectories. To this end, we identify the existence of traps when any change to a variable will cause the resulting Lagrangian value to increase. Using the *hanoi4* and *par16-1* benchmarks, we illustrate that some unsatisfied clauses are trapped more often than others. Since it is too difficult to remember explicitly all the traps encountered, we propose to remember these traps implicitly by giving larger increases to Lagrange multipliers of unsatisfied clauses that are trapped more often. We illustrate the merit of this new update strategy by solving some of most difficult but satisfiable SAT benchmarks in the DIMACS archive (*hanoi4*, *hanoi4-simple*, *par16-1* to *par16-5*, *f2000*, and *par32-1-c* to *par32-3-c*). Finally, we apply the same algorithm to improve on the solutions of some benchmark MAX-SAT problems that we solved before.

Introduction

A general *satisfiability* (SAT) problem is defined as follows. Given a set of n clauses $\{C_1, C_2, \cdots, C_n\}$ on m variables $x = (x_1, x_2, \cdots, x_m)$, $x_j \in \{0, 1\}$, and a Boolean formula in conjunctive normal form:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_n, \tag{1}$$

find a truth assignment to x for (1), where a truth assignment is a combination of variable assignments that makes the Boolean formula true.

The *maximum satisfiability* (MAX-SAT) problem is a general case of SAT. In MAX-SAT, each clause C_i is associated with weight w_i. The objective is to find an assignment of variables that maximizes the sum of weights of satisfied clauses.

Search methods developed previously for solving SAT can be classified into two types. Traditional approaches based on resolution, constraint satisfaction and backtracking are computationally expensive and are not suitable for solving large instances. Local-search methods (Frank 1997; Selman, Kautz, & Cohen 1994; 1993), in contrast, iteratively perturb a trajectory until a satisfiable assignment is found. These methods can solve larger instances, but may have difficulty in solving hard-to-satisfy instances.

Following the successful work of (Shang & Wah 1998), we formulate in this paper a SAT problem as a discrete, constrained optimization problem as follows:

$$\min_{x \in \{0,1\}^m} \quad N(x) = \sum_{i=1}^{n} U_i(x) \tag{2}$$

$$\text{subject to} \quad U_i(x) = 0 \quad \forall i \in \{1, 2, \ldots, n\},$$

where $U_i(x)$ is a binary expression equal to zero when the i^{th} clause is satisfied and to one otherwise, and $N(x)$ is the number of unsatisfied clauses. Note that in the above formulation, when all constraints are satisfied, the objective function is automatically at its minimum.

In this paper, we extend the work of (Shang & Wah 1998; Wu 1998) on discrete Lagrange-multiplier method to solve (2). After summarizing the theory of discrete Lagrange multipliers and the basic approach of (Shang & Wah 1998) for solving SAT problems, we identify traps that limit the search trajectory. Intuitively, *traps* are points in the search space that attract a search trajectory and prevent it from escaping. We present a trap escaping strategy that remembers traps implicitly by increasing the Lagrange multipliers of unsatisfied clauses found in traps, thereby forcing the search not to visit the same traps repeatedly. Finally, we show our results in solving some difficult and previously unsolved satisfiable SAT problems and some MAX-SAT benchmarks in the DIMACS archive.

Research supported by National Science Foundation Grant NSF MIP 96-32316.

Source code of DLM-98 is at http://manip.crhc.uiuc.edu.

Discrete Lagrangian Formulations

(Shang & Wah 1998; Wu 1998) extended the theory of Lagrange multipliers in continuous space to that of discrete space. In contrast to methods in continuous space, Lagrangian methods in discrete space do not require a continuous differentiable space to find equilibrium points. In this section, we summarize the theory of these methods for solving discrete optimization problems. Define a discrete constrained optimization problem as follows:

$$\begin{array}{lll} \min_{x \in E^m} & f(x) & \\ \text{subject to} & g(x) \leq 0 & x = (x_1, x_2, \ldots, x_m) \\ & h(x) = 0 & \end{array} \quad (3)$$

where x is a vector of m discrete variables, $f(x)$ is an objective function, $g(x) = [g_1(x), \ldots, g_k(x)]^T = 0$ is a vector of k inequality constraints, and $h(x) = [h_1(x), \ldots, h_n(x)]^T = 0$ is a vector of n equality constraints.

As discrete Lagrangian methods only handle problems with equality constraints, we first transform an inequality constraint $g_i(x) \leq 0$ into an equality constraint $\max(g_i(x), 0) = 0$. (Shang & Wah 1998) formulates the resulting discrete Lagrangian function as follows:

$$L_d(x, \lambda, \mu) = f(x) + \lambda^T h(x) + \sum_{i=1}^{k} \mu_i \max(0, g_i(x)), \quad (4)$$

where λ and μ are Lagrange multipliers that can be continuous.

The discrete Lagrangian function in (4) cannot be used to derive first-order necessary conditions similar to those in continuous space (Luenberger 1984) because there are no gradients or differentiation in discrete space. Without these concepts, none of the calculus in continuous space is applicable in discrete space.

An understanding of gradients in continuous space shows that they define directions in a small neighborhood in which function values decrease. To this end, (Wu 1998) defines in discrete space a *direction of maximum potential drop* (*DMPD*) for $L_d(x, \lambda, \mu)$ at point x for fixed λ and μ as a vector[1] that points from x to a neighbor of $x \in \mathcal{N}(x)$ with the minimum L_d:

$$\Delta_x L_d(x, \lambda, \mu) = \vec{v}_x = y \ominus x = (y_1 - x_1, \ldots, y_n - x_n) \quad (5)$$

where

$$y \in \mathcal{N}(x) \cup \{x\} \text{ and } L_d(y, \lambda, \mu) = \min_{\substack{x' \in \mathcal{N}(x) \\ \cup \{x\}}} L_d(x', \lambda, \mu). \quad (6)$$

Here, $\ominus$ is the vector-subtraction operator for changing x in discrete space to one of its "user-defined" neighborhood points $\mathcal{N}(x)$. Intuitively, $\vec{v}_x$ is a vector pointing from x to y, the point with the minimum L_d value among all neighboring points of x, including x itself. That is, if x itself has the minimum L_d, then $\vec{v}_x = \vec{0}$.

Based on DMPD, (Shang & Wah 1998; Wu 1998) define the concept of *saddle points* in discrete space similar to those in continuous space (Luenberger 1984). A point (x^*, λ^*, μ^*) is a saddle point when:

$$L(x^*, \lambda, \mu) \leq L(x^*, \lambda^*, \mu^*) \leq L(x, \lambda^*, \mu^*), \quad (7)$$

for all (x^*, λ, μ) and all (x, λ^*, μ^*) sufficiently close to (x^*, λ^*, μ^*). Starting from (7), (Wu 1998) proves stronger first-order necessary and sufficient conditions in discrete space that are satisfied by all saddle points:

$$\Delta_x L_d(x, \lambda, \mu) = 0, \quad \nabla_\lambda L_d(x, \lambda, \mu) = 0, \quad \nabla_\mu L_d(x, \lambda, \mu) = 0. \quad (8)$$

Note that the first condition defines the DMPD of L_d in discrete space of x for fixed λ and μ, whereas the differentiations in the last two conditions are in continuous space of λ and μ for fixed x. Readers can refer to the correctness proofs in (Wu 1998).

The first-order necessary and sufficient conditions in (8) lead to a discrete-space first-order search method that seeks discrete saddle points. The following equations are discrete approximations to implement (8).

General Discrete First-Order Search Method

$$x(k+1) = x(k) \oplus \Delta_x L_d(x(k), \lambda(k), \mu(k)) \quad (9)$$

$$\lambda(k+1) = \lambda(k) + c_1 h(x(k)) \quad (10)$$

$$\mu(k+1) = \mu(k) + c_2 \max(0, g(x(k))) \quad (11)$$

where $\oplus$ is the vector-addition operator ($x \oplus y = (x_1 + y_1, \ldots x_n + y_n)$), and c_1 and c_2 are positive real numbers controlling how fast the Lagrange multipliers change.

It is easy to see that the necessary condition for (9)-(11) to converge is when $h(x) = 0$ and $g(x) \leq 0$, implying that x is a feasible solution to the original problem. If any of the constraints is not satisfied, then λ and μ on the unsatisfied constraints will continue to evolve. Note that, as in continuous Lagrangian methods, the first-order conditions are only satisfied at saddle points, but do not imply that the time to find a saddle point is finite, even if one exists.

DLM for Solving SAT Problems

DLM-98-BASIC-SAT: A Basic DLM

The advantage of formulating the solution of SAT as discrete Lagrangian search is that the method has a solid mathematical foundation (Shang & Wah 1998; Wu 1998). The theory of discrete Lagrange multipliers also explains why other weight-update heuristics (Frank 1997; Morris 1993) work in practice, although these heuristics were developed in an ad hoc fashion.

[1] To simplify our symbols, we represent points in the x space without the explicit vector notation.

procedure *DLM-98-BASIC-SAT*
1. Reduce original SAT problem;
2. Generate a random starting point using a fixed seed;
3. Initialize $\lambda_i \longleftarrow 0$;
4. **while** solution not found and time not used up **do**
5. Pick $x_j \notin$ TabuList that reduces L_d the most;
6. Maintain TabuList;
7. Flip x_j;
8. **if** $\#_{UpHillMoves} + \#_{FlatMoves} > \theta_1$ **then**
9. $\lambda_i \longleftarrow \lambda_i + \delta_o$;
10. **if** $\#_{Adjust}\% \ \theta_2 = 0$ **then**
11. $\lambda_i \longleftarrow \lambda_i - \delta_d$ **end_if**
12. **end_if**
13. **end_while**

end

Figure 1: *DLM-98-BASIC-SAT* (Shang & Wah 1998): An implementation of the basic discrete first-order method for solving SAT.

The Lagrangian function for the SAT problem in (2) is:

$$L_d(x, \lambda) = N(x) + \sum_{i=1}^{n} \lambda_i U_i(x) \qquad (12)$$

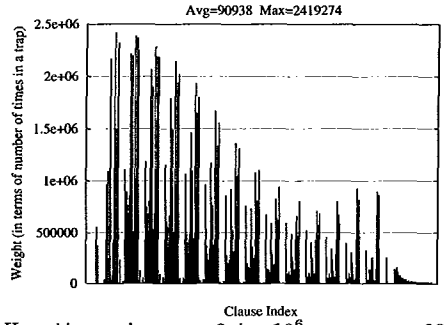
Figure 1 shows the basic *Discrete Lagrangian Method* (*DLM*) of (Shang & Wah 1998) for solving SAT problems. It uses two heuristics, one based on tabu lists (Glover 1989) and the other based on flat (Selman, Kautz, & Cohen 1993) and up-hill moves. We explain these steps later when we present our improved DLM.

Although quite simple, DLM-98-BASIC-SAT can find solutions to most satisfiable DIMACS benchmarks, such as all problems in the *aim-*, *ii-*, *jnh-*, *par8-*, *ssa-* classes, within seconds. However, it takes a long time to solve some DIMACS benchmarks and has difficulty in solving a few of the large ones (Shang & Wah 1998). For example, it takes a long time to solve *f2000* and *par16-1-c* to *par16-5-c* and cannot solve *hanoi4*, *hanoi4-simple*, *hanoi5*, *par16-1* to *par16-5*, and all *par32-* problems.

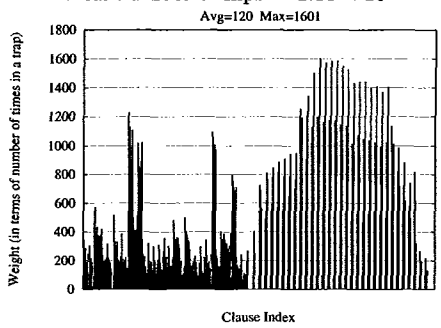
To improve DLM-98-BASIC-SAT, we identify in the next subsection traps that prevent DLM trajectories from moving closer to satisfiable assignments. We then propose new strategies to overcome these traps.

Traps to Local Search

By examining the output profiles when applying DLM-98-BASIC-SAT to solve hard SAT problems, we find that some clauses are frequently flipped from being satisfied to being unsatisfied. A typical scenario is as follows. A clause is initially unsatisfied but becomes satisfied after a few flips due to increases of λ for that clause. It then becomes unsatisfied again after a few more flips due to increases of λ of other unsatisfied clauses. These state changes happen repeatedly for some clauses and are tremendously inefficient because they trap the trajectory in an unsatisfiable assignment. To quantify the observations, we introduce a new concept called traps.



(a) *Hanoi4*: maximum = 2.4×10^6, average = 90,938, total number of flips = 1.11×10^8



(b) *Par-16-1*: maximum = 1.6×10^3, average = 120, total number of flips = 5×10^6

Figure 2: Large disparity between the maximum and average numbers of times a clause is in traps.

A *trap* is a combination of x and λ such that a point in it has one or more unsatisfied clauses, and any change to a single variable in x will cause the associated L_d to increase. Note that a satisfiable assignment is not a trap because all its clauses are satisfied, even though its L_d may increase when x is perturbed.

To show that some clauses are more likely to be unsatisfied, we plot the number of times a clause is in a trap. This is not the same as the number of times a clause is unsatisfied because a clause may be unsatisfied when outside a trap. We do not consider the path a search takes to reach a trap, during which a clause may be unsatisfied, because the different paths to reach a trap are not crucial in determining the strategy to escape from it.

Figure 2 shows that some clauses reside in traps much more often than average when *DLM-98-BASIC-SAT* was applied to solve *hanoi4* and *par16-1*, two very hard SAT problems in the DIMACS archive. This behavior is detrimental to finding solutions because the search may be trapped at some points for a long time, and the search is restricted to a small area in the search space.

Ideally, we like a trajectory to never visit the same point twice in solving an optimization problem. This is, however, difficult to achieve in practice because it

procedure *DLM-99-SAT*
1. Reduce original SAT problem;
2. Generate a random starting point using a fixed seed;
3. Initialize $\lambda_i \longleftarrow 0$ and $t_i \longleftarrow 0$;
4. **while** solution not found and time not used up **do**
5. Pick $x_j \notin$ TabuList that reduces L_d the most;
6. **If** search is in a trap **then**
7. For all unsatisfied clauses u, $t_u \longleftarrow t_u + \delta_w$ **end_if**
8. Maintain TabuList;
9. Flip x_j;
10. **if** $\#_{UpHillMoves} + \#_{FlatMoves} > \theta_1$ **then**
11. $\lambda_i \longleftarrow \lambda_i + \delta_o$;
12. **if** $\#_{Adjust} \% \theta_2 = 0$ **then**
13. $\lambda_i \longleftarrow \lambda_i - \delta_d$; **end_if**;
14. call *SPECIAL-INCREASE*;
15 **end_if**
16. **end_while**

end

procedure *SPECIAL-INCREASE*
17. Pick a set of clauses S;
18. **if** $\frac{\max_{i \in S} t_i}{\sum_{i \in S} t_i / n} \geq \theta_3$ **then**
19. For clause i in S with the largest t_i, $\lambda_i \longleftarrow \lambda_i + \delta_s$;
20. **end_if**

end

Figure 3: Procedures *DLM-99-SAT*, an implementation of the discrete first-order method for solving SAT problems, and *SPECIAL-INCREASE*, special increments of λ on certain clauses when their weights are out of balance.

is impractical to keep track of the history of an entire trajectory. Alternatively, we can try not to repeat visiting the same trap many times. The design of such a strategy will depend on how we escape from traps.

There are three ways to bring a trajectory out of traps; the first two maintains history information explicitly, while the last maintains history implicitly.

a) We can perturb two or more variables at a time to see if L_d decreases, since a trap is defined with respect to the perturbation of one variable. This is not practical because there are too many combinations to enumerate when the number of variables is large.

b) We can restart the search from a random starting point in another region when it reaches a trap. This will lose valuable history information accumulated during each local search and is detrimental in solving hard SAT problems. Moreover, the history information in each local search needs to be maintained explicitly.

c) We can update λ to help escape from a trap. By placing extra penalties on all unsatisfied clauses inside a trap, unsatisfied clauses that are trapped more often will have very large Lagrange multipliers, making them less likely to be unsatisfied in the future. This strategy, therefore, implicitly reduces the probability of visiting that same trap again in the future and was used in our experiments.

DLM-99-SAT: An Improved DLM for SAT

Figure 3 outlines the new DLM for solving SAT. It defines a weight for each Lagrange multiplier and increases the weights of all unsatisfied clauses every time the search reaches a trap. This may, however, lead to an undesirable out-of-balance situation in which some clauses have much larger weights than average. To cope with this problem, when the ratio of the largest weight to the average is larger than a predefined threshold, we increase the Lagrange multipliers of clauses with the largest weight in order to force them into satisfaction. If these increases are large enough, the corresponding unsatisfied clauses are not likely to be unsatisfied again in future, thereby resolving the out-of-balance situation.

We explain next in detail each line of DLM-99-SAT.

Line 1 carries out straightforward reductions on all one-variable clauses. For all one-variable clauses, we set the value of that variable to make that clause satisfied and propagate the assignment.

Line 2 generates a random starting point using a fixed seed. This allows the experiments to be repeatable.

Line 3 initializes t_i (temporary weight for Clause i) and λ_i (Lagrange multiplier for Clause i) to zero in order to make the experiments repeatable. Note that t_i increases λ_i faster if it is larger.

Line 4 defines a loop that will stop when time (maximum number of flips) runs out or when a satisfiable assignment is found.

Line 5 chooses a variable x_j that will reduce L_d the most among all variables not in *TabuList*. If such cannot be found, then it picks x_j that will increase L_d the least. We call a flip an *up-hill move* if it causes L_d to increase, and a *flat move* (Selman, Kautz, & Cohen 1993) if it does not change L_d. We allow flat and up-hill moves to help the trajectory escape from traps.

Lines 6-7 locate a trap and increase t_u by δ_w $(= 1)$ for all unsatisfied clauses in that trap.

Line 8 maintains *TabuList*, a first-in-first-out queue with a problem-dependent length *tabu_len* (100 for f, 10 for *par*16 and *par*32, 16 for g, and 18 for *hanoi*4).

Line 9 flips the x_j chosen (from false to true or vice versa). It also records the number of times the trajectory is doing flat and up-hill moves.

Lines 10-11 increase the Lagrange multipliers for all unsatisfied clauses by δ_o $(= 1)$ when the sum of up-hill and flat moves exceeds a predefined threshold θ_1 (50 for f, 16 for *par*16 and *par*32, 26 for g, and 18 for *hanoi*4). Note that δ_o is the same as c_1 in (10). After increasing the Lagrange multipliers of all unsatisfied clauses, we increase a counter $\#_{Adjust}$ by one.

Lines 12-13 reduce the Lagrange multipliers of all clauses by δ_d $(= 1)$ when $\#_{Adjust}$ reaches threshold θ_2 (12 for f, 46 for *par*16, 56 for *par*32, 6 for g, and 40

for *hanio4*). These help change the relative weights of all the clauses and may allow the trajectory to go to another region in the search space after the reduction.

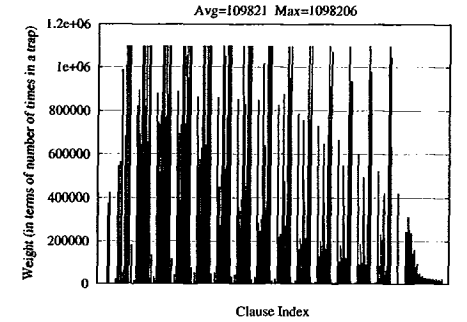
Line 14 calls Procedure *SPECIAL-INCREASE* to handle the case when some clauses appear in traps more often than other clauses.

Line 17 picks a problem-dependent set S of clauses (for *par16-1* to *par16-5*, the set of all currently unsatisfied clauses; for others, the set of all clauses).

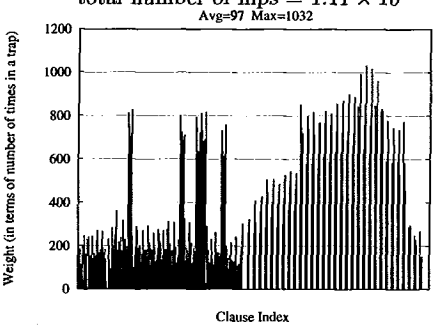
Lines 18-19 compute the ratio between the maximum weight and the average weight to see if the ratio is out of balance, where n is the number of clauses. If the ratio is larger than θ_3 (3 for *par*16, *par*32, and f, 1 for g, and 10 for *hanoi*4), then we increase the Lagrange multiplier of the clause with the largest weight by δ_s (1 for all problems).

Intuitively, increasing the Lagrange multipliers of unsatisfied clauses in traps can reduce their chance to be in traps again. Figure 4 illustrates this point by plotting the number of times that clauses appear in traps after using *SPECIAL-INCREASE*. Compared to Figure 2, we see that *SPECIAL-INCREASE* has controlled the large imbalance in the number of times that clauses are unsatisfied. For *hanoi4* (resp. *par16-1*), the maximum number of times a clause is trapped is reduced by more than 50% (resp. 35%) after the same number of flips.

Note that the balance is controlled by parameters θ_3 and δ_s. If we use smaller θ_3 and larger δ_s, then better balance can be achieved. However, better balance does not always lead to better solutions because a search may leave a trap quickly using smaller θ_3 and larger δ_s, thereby missing some solutions for hard problems.



(a) *Hanoi4*: maximum $= 1.1 \times 10^6$, average $= 109,821$, total number of flips $= 1.11 \times 10^8$



(b) *Par16-1*: maximum $= 1,032$, average $= 97$, total number of flips $= 5 \times 10^6$

Figure 4: Reduced disparity between the maximum and average numbers of times a clause is in traps using *SPECIAL-INCREASE*.

Results on SAT and MAX-SAT

We first apply DLM-99-SAT to solve some hard but satisfiable SAT problems in the DIMACS archive. DLM-99-SAT can now solve quickly *f2000*, *par16-1-c* to *par16-5-c*, *par16-1* to *par16-5*, and *hanoi4* with a very high success ratio. These problems had not been solved well by any single method in the literature. Moreover, it can now solve *hanoi4-simple* with a very high success ratio, and *par32-1-c* to *par32-3-c*, although not with high success ratios. These problems cannot be solved by any other local search method today. For other simpler problems in the DIMACS archive, DLM-99-SAT has similar performance as the best existing method developed in the past. Due to space limitation, we will not present the details of these experiments here.

Table 1 lists the experimental results on all the hard problems solved by DLM-99-SAT and the experimental results from WalkSAT and GSAT. It lists the CPU times of our current implementation on a Pentium-Pro 200 MHz Linux computer, the number of (machine independent) flips for our algorithm to find a feasible solution, the success ratios (from multiple randomly generated starting points), and in the last two columns the success ratios (SR) and CPU times of WalkSAT/GSAT. For most problems, we tried our algorithm from 10 random starting points. For *hanoi4* and *hanoi4-simple*, we only tried 6 starting points because each run took more than 50 hours of CPU time on the average to complete. Note that *hanoi4*, *hanoi4-simple* and *par32-* problems are much harder than problems in the *par16* and *f* classes because the number of flips is much larger.

Table 1 also lists the results of applying DLM-99-SAT to solve the g-class problems that were not solved well by (Shang & Wah 1998). The number of flips used for solving these problems indicate that they are much easier than problems in the *par16* class.

So far, we are not able to find solutions to eight satisfiable problems in the DIMACS archive (*hanoi5*, *par32-4-c* to *par32-5-c* and *par32-1* to *par32-5*). However, we have found assignments to *hanoi5* and *par32-2-c* to *par32-5-c* with only one unsatisfied clause. These results are very encouraging from the point view of the number of unsatisfied clauses.

Next, we apply the same algorithm to improve on the solutions of some MAX-SAT benchmark problems

Table 1: Comparison of performance of DLM-99-SAT for solving some hard SAT problems and the g-class problems that (Shang & Wah 1998) did not solve well before. (All our experiments were run on a Pentinum Pro 200 computer with Linux. WalkSAT/GSAT experiments were run on an SGI Challenge with MPIS processor, model unknown. "NA" in the last two columns stands for "not available.")

Problem ID	Succ. Ratio	CPU Sec.	Num. of Flips	WalkSAT/GSAT SR	WalkSAT/GSAT Sec.
par16-1	10/10	216.5	$1.3 \cdot 10^7$	NA	NA
par16-2	10/10	406.3	$2.7 \cdot 10^7$	NA	NA
par16-3	10/10	309.2	$2.1 \cdot 10^7$	NA	NA
par16-4	10/10	174.8	$1.2 \cdot 10^7$	NA	NA
par16-5	10/10	293.6	$2.0 \cdot 10^7$	NA	NA
par16-1-c	10/10	79.9	5501464	NA	NA
par16-2-c	10/10	124.4	8362374	NA	NA
par16-3-c	10/10	116.0	7934451	NA	NA
par16-4-c	10/10	111.3	7717847	NA	NA
par16-5-c	10/10	81.9	5586538	NA	NA
f600	10/10	1.4	39935	NA	35*
f1000	10/10	8.3	217061	NA	1095*
f2000	10/10	44.3	655100	NA	3255*
hanoi4	5/6	$1.85 \cdot 10^5$	$4.7 \cdot 10^9$	NA	NA
hanoi4_s	5/6	$2.58 \cdot 10^5$	$9.9 \cdot 10^9$	NA	NA
par32-1-c	1/10	$5.36 \cdot 10^4$	6411650	NA	NA
par32-2-c	1/20	$2.16 \cdot 10^5$	$9.2 \cdot 10^9$	NA	NA
par32-3-c	1/30	$3.27 \cdot 10^5$	$1.4 \cdot 10^{10}$	NA	NA
g125-17	10/10	231.5	632023	7/10**	264**
g125-18	10/10	10.9	8805	10/10**	1.9**
g250-15	10/10	25.6	2384	10/10**	4.41**
g250-29	10/10	412.1	209813	9/10**	1219**

*: Results from (Selman, Kautz, & Cohen 1993) for similar but not the same problems in the DIMACS archive
**: Results from (Selman 1995)

solved by (Shang & Wah 1997) before. Recall the weight on Clause i is w_i, and the goal is to maximize the weighted sum of satisfied clauses. In solving MAX-SAT, we set initial values in DLM-99-SAT (Figure 3) to be $\theta_1 \leftarrow 20$, $\theta_2 \leftarrow 74$, $\theta_3 \leftarrow 10$ and for Clause i, $\lambda_i \leftarrow w_i + 1$, $\delta_o \leftarrow 2w_i$, $\delta_d \leftarrow w_i/4$, and $\delta_s \leftarrow 5w_i/4$.

Using these empirically set parameters, we were able to find optimal solutions to all the MAX-SAT benchmark problems within 20 runs and 10,000 flips. Table 2 presents the results with respect to the number of successes from 20 randomly generated starting points and the average CPU seconds when optimal solutions were found. For cases that did not lead to optimal solutions, we show in the third column the average deviation from the optimal solutions. The last column shows the indices of the *jnh* problems in the MAX-SAT benchmarks achieving the results.

Our algorithm solves MAX-SAT better than (Shang & Wah 1997) and GRASP, but our average number of successes of 14.77 is slightly worse than the average of 16.64 in (Mills & Tsang 1999). This could be due to the fact that our algorithm was originally designed for solving SAT rather than MAX-SAT.

Table 2: Performance of DLM-99-SAT in solving the 44 MAX-SAT DIMACS benchmark problems (Shang & Wah 1997). (Each problem were solved on a Sun Ultra-5 computer from 20 randomly generated starting points and with a limit of 10,000 flips. CPU sec. is the average CPU time for runs that led to the optimal solution. Deviation from optimal solution is the average deviation for runs that did not lead to the optimal solution.)

# of Succ.	CPU Sec.	Deviation from Opt.	List of *jnh*–? Benchmark Problems Achieving Performance
2	0.06	-175.2	9
4	0.07	-33.0	18
5	0.12	-78.0	4, 5, 11, 303, 305, 307
6	0.11	-83.4	15
9	0.11	-29.4	310
10	0.13	-2.8	16
12	0.09	-47.4	19, 208
14	0.11	-49.3	302
15	0.10	-15.6	215, 216, 219, 309
16	0.11	-13.9	6, 8, 203
17	0.11	-19.3	211, 212, 308
18	0.08	-3.5	14, 207, 220
19	0.07	-1.1	301, 304
20	0.03	0.0	1, 7, 10, 12, 13, 17, 201, 202, 205, 209, 210, 214, 217, 218, 306

References

Frank, J. 1997. Learning short-term weights for GSAT. *Proc. 15'th Int'l Joint Conf. on AI* 384–391.

Glover, F. 1989. Tabu search — Part I. *ORSA J. Computing* 1(3):190–206.

Luenberger, D. G. 1984. *Linear and Nonlinear Programming.* Addison-Wesley Publishing Company.

Mills, P., and Tsang, E. 1999. Solving the MAX-SAT problem using guided local search. *Technical Report CSM-327, University of Essex, Colchester, UK.*

Morris, P. 1993. The breakout method for escaping from local minima. In *Proc. of the 11th National Conf. on Artificial Intelligence*, 40–45.

Selman, B.; Kautz, H.; and Cohen, B. 1993. Local search strategies for satisfiability testing. In *Proc. of 2nd DIMACS Challenge Workshop on Cliques, Coloring, and Satisfiability, Rutgers University*, 290–295.

Selman, B.; Kautz, H.; and Cohen, B. 1994. Noise strategies for improving local search. In *Proc. of 12th National Conf. on Artificial Intelligence*, 337–343.

Selman, B. 1995. Private communcation.

Shang, Y., and Wah, B. W. 1997. Discrete lagrangian-based search for solving MAX-SAT problems. *Proc. 15'th Int'l Joint Conf. on AI* 378–383.

Shang, Y., and Wah, B. W. 1998. A discrete Lagrangian based global search method for solving satisfiability problems. *J. Global Optimization* 12(1):61–99.

Wu, Z. 1998. *Discrete Lagrangian Methods for Solving Nonlinear Distrete Constrained Optimization Problems.* Urbana, IL: M.Sc. Thesis, Dept. of Computer Science, Univ. of Illinois.