An Efficient Global-Search Strategy in Discrete Lagrangian Methods for Solving Hard Satisfiability Problems

Zhe Wu and Benjamin W. Wah
Department of Electrical and Computer Engineering
and the Coordinated Science Laboratory
University of Illinois at 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 an efficient global-search strategy in an algorithm based on the theory of discrete Lagrange multipliers for solving difficult SAT instances. These difficult benchmarks generally have many traps and basins that attract local-search trajectories. In contrast to trap-escaping strategies proposed earlier (Wu & Wah 1999a; 1999b) that only focus on traps, we propose a global-search strategy that penalizes a search for visiting points close to points visited before in the trajectory, where penalties are computed based on the Hamming distances between the current and historical points in the trajectory. The new strategy specializes to the earlier trap-escaping strategies because when a trajectory is inside a trap, its historical information will contain many points in close vicinity to each other. It is, however, more general than trap escaping because it tries to avoid visiting the same region repeatedly even when the trajectory is not inside a trap. By comparing our results to existing results in the area (Wu & Wah 1999a; 1999b; Kautz & Selman 1996; Choi, Lee, & Stuckey 1998; Marques-Silva & Sakalla 1999), we conclude that our proposed strategy is both effective and general.

Introduction

Satisfiability (SAT) problems are the most fundamental discrete, constraint-satisfaction problems among all NP-complete problems. Many real-world applications, like artificial intelligence, computer-aided design, database processing, and planning, can be formulated as SAT problems. These problems generally require algorithms of exponential complexity in the worst case in order to obtain satisfiable assignments.

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

\[
C_1 \land C_2 \land \cdots \land C_n, \tag{1}
\]

find a truth assignment to $x$ for (1), where a truth assignment is a combination of variable assignments that makes the Boolean formula true. For example, one solution to the following simple SAT problem with four variables and four clauses is $(1, 0, 0, 0)$:

\[
(x_1 \lor x_3 \lor x_4) \land (x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_2 \lor x_4) \land (x_1 \lor x_2 \lor x_3). \tag{2}
\]

Many algorithms and heuristics have been developed to solve SAT problems. Complete approaches based on resolution, constraint satisfaction and backtracking (Pudim 1983; Genesereth & Nilsson 1987; Davis & Putnam 1960) are computationally expensive and are not suitable for solving large instances. Local-search methods (Frank 1997; Selman, Kautz, & Cohen 1994; 1993; Gu 1993; Sosic & Gu 1994), 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. Recently, there is also some research in combining global optimization schemes, like GA, with local-search methods (Polino, Pizzuti, & Spezzano 1998).

In this paper, we formulate a SAT problem as a discrete, constrained optimization problem as follows:

\[
\min_{x \in \{0, 1\}^n} \quad N(x) = \sum_{i=1}^{n} U_i(x)
\]

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

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. In this formulation, the objective function is automatically at its minimum when all the clauses are satisfied. Although the redundancy may seem superficial, it actually gives better control in constraint satisfaction by treating each clause as a separate constraint. In contrast, in an unconstrained formulation, all clauses may be viewed to have the same fixed relative weight. This is generally undesirable because different clauses may have different degrees of difficulty to be satisfied. Hence, a constrained formulation of SAT with dynamically changing penalties on each clause is intuitively preferable.
In this paper, we briefly introduce the theory of discrete Lagrange multipliers and the discrete Lagrange-multiplier method (DLM) that can be applied to solve (2) (Shang & Wah 1998). We then show a basic implementation of DLM used in (Shang & Wah 1998) for solving SAT problems and discuss reasons why some hard instances cannot be solved. To solve those difficult instances, we propose a global-search strategy that avoids visiting nearby points visited before, using penalties related to the Hamming distances between the current and the historical points in the search trajectory. A global-search strategy in this paper is defined as one that can overcome local minima or valleys in the search space under consideration. It differs from global-optimization strategies in the sense that there is no theoretical guarantee that it will converge to a feasible or an optimal solution even when sufficient time is given. Finally, we show our results in solving some difficult SAT benchmark instances in the DIMACS archive and compare our results to existing results in this area.

**Discrete Lagrangian Formulations**

In this section, we summarize briefly the theory of discrete Lagrange multipliers (Shang & Wah 1998; Wu 1998; Wah & Wu 1999) for solving general constrained discrete optimization problems.

Define a discrete constrained optimization problem as:

\[ \min_{x \in \mathcal{E}^m} f(x) \quad \text{subject to} \quad h(x) = 0, \quad x = (x_1, x_2, \ldots, x_m), \]

where \( x \) is a vector of \( m \) discrete variables, \( f(x) \) is an objective function, and \( h(x) = [h_1(x), \ldots, h_n(x)]^T = 0 \) is a vector of \( n \) equality constraints. The corresponding discrete Lagrangian function is defined as follows:

\[ L_d(x, \lambda) = f(x) + \lambda^T h(x), \]

where \( \lambda \) is a vector of Lagrange multipliers that can be either continuous or discrete.

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; Wah & Wu 1999) define in discrete space a direction of maximum potential drop (DMPD) for \( L_d(x, \lambda) \) at point \( x \) for fixed \( \lambda \) as a vector that points from \( x \) to a neighbor of \( x \in \mathcal{N}(x) \) with the minimum \( L_d \):

\[ \Delta_x L_d(x, \lambda) = \Delta_x = y \Theta x = (y_1 - x_1, \ldots, y_n - x_n) \quad \text{where} \quad y \in \mathcal{N}(x) \cup \{x\} \text{ and } L_d(y, \lambda) = \min_{x' \in \mathcal{N}(x)} L_d(x', \lambda). \]

Here, \( \Theta \) is the vector-subtraction operator for changing \( x \) in discrete space to one of its "user-defined" neighborhood

\[ \Delta_x = \mathbf{0}. \]

Intuitively, \( \Delta_x \) 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 \( \Delta_x = \mathbf{0} \).

Based on DMPD and a concept called saddle points introduced in (Shang & Wah 1998), (Wu 1998; Wah & Wu 1999) prove some first-order necessary and sufficient conditions on discrete-space constrained local minima based on the minimum of the discrete Lagrangian function defined in (4). This is done by showing that the first-order conditions are necessary and sufficient for a point to be a discrete saddle point, and that the saddle-point condition is necessary and sufficient for a point to be a constrained local minimum. Readers should refer to the correctness proofs in (Wu 1998).

The first-order conditions lead to the following iterative procedure to look for solutions to (3):

**General Discrete First-Order Search Method**

\[ x(k+1) = x(k) \Theta \Delta_x L_d(x(k), \lambda(k)) \quad (7) \]

\[ \lambda(k+1) = \lambda(k) + \Delta \lambda(x(k)) \quad (8) \]

where \( \Theta \) is the vector-addition operator (\( x \Theta y = (x_1 + y_1, \ldots, x_n + y_n) \)), and \( \Delta \) is a positive real number controlling how fast the Lagrange multipliers change.

It is easy to see that the necessary condition for (7) to converge is when \( h(x) = 0 \), implying that \( x \) is a feasible solution to the original problem. If any of the constraints is not satisfied, then \( \lambda \) on the unsatisfied constraints will continue to evolve until the constraint is satisfied. Note that, similar to continuous Lagrangian methods, there is no guarantee for the first-order method to find a feasible solution in finite time, even if one exists.

**Basic DLM Implementation for SAT**

We describe the solution of SAT as a discrete Lagrangian search in this section. Although the overall strategy for updating Lagrange multipliers may resemble existing weight-update heuristics (Frank 1997; Morris 1993), our proposed formulation is based on a solid mathematical foundation of discrete Lagrange multipliers. The Lagrangian search, when augmented by new heuristics presented in the next section, provides a powerful tool to solve hard-to-satisfy SAT instances.

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

\[ L_d(x, \lambda) = N(x) + \sum_{i=1}^{n} \lambda_i U_i(x) \quad (9) \]

where \( U_i(x) \) is a binary function equal to zero when the \( i \)-th clause is satisfied and to one otherwise, and \( N(x) \) is the number of unsatisfied clauses.
procedure DLM-98-BASIC-SAT
1. Reduce the original SAT problem;
2. Generate a random starting point using a fixed seed;
3. Initialize λ₁ ← 0;
4. while solution not found and time not used up do
5. Pick xₙ for TabuList that reduces L_d the most;
6. Maintain TabuList;
7. Flip xₙ;
8. if #FlatMoves > θ₁ then
9. λ₁ ← λ₁ + δ₁;
10. if #AdjAbs δ₁ = 0 then
11. λ₁ ← λ₁ - δ₁ 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 (7) and (8) for solving SAT.

Figure 1 shows a basic implementation of discrete Lagrangian method (DLM) of (Shang & Wah 1998). The original DLM uses two main heuristics: tabu lists (Glover 1989) and flat moves (Selman, Kautz, & Cohen 1993). We explain the steps of DLM later when we present our proposed global-search strategy.

Table 1 lists the performance of our current implementation of DLM on a 500-MHz Pentium-III computer with Solaris 7 (from 10 randomly generated starting points) for some typical DIMACS/SATLIB benchmark problems. Due to space limitation, we only present our results for a few representative instances in each class of problems.

Although quite simple, DLM-98-BASIC-SAT can find solutions within seconds to most satisfiable DIMACS benchmarks, such as all the problems in the aim, ii, jnh, par8, and ssa classes, and most problems in SATLIB, like uniform 3-SAT problems uf, flat graph coloring flat, and morphed graph coloring sw problems. DLM-98-BASIC-SAT is either faster than, or at least comparable to, competing algorithms like GSAT and Grasp (Shang & Wah 1998). However, it has difficulty in solving problems in the par16-, hanoi-, g-, f2000- and par32-classes.

DLM with Global Search for SAT

Traps were identified by (Wu & Wah 1999b; 1999a) as one of the major difficulties in applying DLM to solve hard SAT instances. A trap is a basin in the original variables, x, of a discrete Lagrangian space in which a flip of any variable inside the trap will only cause the Lagrangian value to increase. In other words, there is no viable descents in x space through a single flip, and the trajectory is stuck until enough changes are made to the Lagrangian multipliers. Since changing the Lagrange multipliers on unsatisfied clauses changes the terrain, a trap will disappear eventually when the multipliers of unsatisfied clauses are large enough. Traps degrade the performance of DLM because many cycles are spent in updating Lagrange multipliers gradually.

A trap-escaping strategy was proposed in (Wu & Wah 1999b; 1999a) to overcome the attraction of traps in DLM.
Its basic idea is to keep track of clauses appearing in traps and add extra penalties to those clauses by increasing their corresponding Lagrange multipliers. Although the strategy was tested to be very helpful in overcoming traps and in finding better solutions, it only corrects the search after it gets into the same trap repeatedly.

A more general way is to avoid visiting the same set of points visited in the past. This is done by measuring the distances in $x$ space between the current point and points visited previously in the trajectory and by penalizing the search accordingly. This strategy leads to an efficient global-search method that addresses not only the repulsion of traps in $x$ space but also the repulsion of points visited in the past.

The new discrete Lagrangian function, including the extra distance penalty term, is defined as:

$$L_d(x, \lambda) = N(x) + \sum_{i=1}^{n} \lambda_i U_i(x) - \text{distance penalty}.$$  \hspace{1cm} (10)

Distance penalty is actually the sum of Hamming distances from the current point to some points visited in the past in the trajectory. Hence, if a trajectory is stuck in a trap, then the Hamming distances from the current point to points in the trap will be small, leading to a large penalty on the Lagrangian function (10). On the other hand, if the trajectory is not inside a trap, then the Hamming distance to points close to points visited before will still be large, thereby steering the trajectory away from regions visited before.

The exact form of distance penalty is defined to be:

$$\text{distance penalty} = \sum_{i} \min(\theta_i, |x - x_i^e|),$$  \hspace{1cm} (11)

where $\theta_i$ is a positive threshold, and $|x - x_i^e|$ is the Hamming distance between point $x$ to a point $x_i^e$ visited before by the trajectory.

$\theta_i$ is used to control the search and put a lower bound on distance penalty so that it will not be a dominant factor in the new Lagrangian function. Without $\theta_i$, the first-order search method will prefer a far-away point than a point with less constraint violation, which is not desirable. In our experiments, we set $\theta_i$ to be 2. This means that, when the Hamming distances between the current point and all the stored historical points of a trajectory is larger than 2, the impact of all the stored historical points on distance penalty will be the same. Note that the fundamental concept of distance penalty is similar to that of tabu search (Glover 1989).

Ideally, we like to store all the points traversed by a trajectory in the past. This is, however, impractical in terms of memory usage and computation overhead in calculating DMPD for the ever-increasing Lagrangian value. In our implementation, we keep a fixed-size queue of size $q_a$ of historical points and periodically update this queue in a FIFO manner. The period of update is based on $w_a$ flips; namely, after $w_a$ flips, we save the current search point in the queue and remove the oldest element from the queue.

**DLM-2000-SAT**

Figure 2 shows the algorithm of DLM-2000 for solving general SAT instances. In the following, we explain each line of the algorithm in detail, including the various parameters and their values. In general, we need a unique set of parameters for each class of problems in order to achieve the best performance. For the around 200 instances in the DIMACS and SATLIB benchmarks, we need only five different sets of parameters.

Line 1 performs some straightforward reductions on all clauses with a single variable. For all single-variable clauses, we set the value of that variable to make that clause satisfied and propagate the assignment. For example, if a clause has just one variable $x_a$, then $x_a$ must be true in the assignment of the solution. Reduction stops when there are no single-variable clauses.

Line 2 generates a random starting point using a fixed seed. Note that we use the long-period random-number generator of L’Ecuyer with Bays-Durham shuffle and added safeguards rather than the default generator provided in the C library in order to allow our results to be reproducible across different platforms.

Line 3 initializes $\lambda$ (Lagrange multiplier for Clause $i$) to zero in order to make the experiments repeatable.

Line 4 is the main loop of our algorithm that stops 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 a variable cannot be found, then it picks $x_j$ that will not increase $L_d$. We call a flip a flat move (Selman, Kautz, & Cohen 1993) if it does not change $L_d$. We allow flat moves to help the trajectory explore flat regions.

**Figure 2**: Procedures DLM-2000-SAT, an implementation of the discrete first-order method for solving SAT problems.
Line 6 flips the \( x_j \) chosen (from false to true or vice versa). It also records the number of times the trajectory is doing flat moves.

Lines 7-8 maintain a queue on a fixed number of historical points. After a predefined number of flips, the algorithm stores the current search point in the queue and removes the oldest historical point. Note that this queue needs to be designed carefully in order to make the whole scheme efficient. In our experiments, we choose the queue size \( q \) to be in the range \([4, 20] \).

Line 9 maintains TabuList. Similar to the queue for storing historical points, TabuList is also a FIFO queue. Each time a variable is flipped, it will be put in TabuList, and the oldest element will be removed from TabuList. TabuLists are important in helping a search explore flat regions effectively.

Lines 10-11 increase the Lagrange multipliers for all unsatisfied clauses by \( \delta = 1 \) when the number of flat moves exceeds a predefined threshold \( \delta = 50 \) for \( f \), 16 for \( par16 \), 36 for \( g \), and 16 for \( hanoi4 \). Note that \( \delta = 1 \) is the same as \( c_i \) in (8). After increasing the Lagrange multipliers of all unsatisfied clauses, we increase counter \# Adjust by one.

Lines 12-13 reduce the Lagrange multipliers of all clauses by \( \delta = 1 \) when \# Adjust reaches threshold \( \delta = 12 \) for \( f \), 46 for \( par16 \), 7 for \( g \), and 40 for \( hanoi4 \). These help change the relative weights of all the clauses and may allow the trajectory to visit another region in the search space after the reduction. They are critical to our global-search strategy because they help maintain the effect of distance penalty in the Lagrangian function. Their purpose is to keep the weighted constraint functions, \( \lambda^T h(x) \), in the Lagrangian definition to be in a suitable range, given that distance penalty has a fixed range that can be computed from \( w \) and \( \delta \). Otherwise, when \( \lambda \) gets too large, distance penalty will be relatively small and has no serious effect in avoiding regions visited before.

Compared to DLM-99-SAT (Wu & Wah 1999b; 1999a), our proposed algorithm is simpler and has less parameters to be tuned. Note that there is more overhead in searching for a suitable variable to flip (Line 5); that is, each flip will take more CPU time than a similar flip in DLM-99-SAT. However, the overall CPU time is actually much shorter for most benchmark problems tested because the new global-search strategy can avoid visiting the same regions more effectively.

### Results on Some Hard SAT Instances

We first apply DLM-2000-SAT to solve some hard, satisfiable SAT instances in the DIMACS archive. DLM-2000-SAT can now solve quickly \( f2000 \), \( par16-1-c \) to \( par16-5-c \), \( par16-1 \) to \( par16-5 \), \( hanoi4 \) and \( hanoi4 \)-simple with 100% success ratio. For other simpler problems in the DIMACS archive, DLM-2000-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 2 lists the experimental results on all the hard problems solved by DLM-2000-SAT, WalkSAT, GSAT, and DLM-99-SAT. It lists the CPU times of our implementation on a 500-MHz Pentium-III computer with Solaris 7, the numbers of (machine-independent) flips for our algorithm to find a feasible solution, the success ratios (from multiple randomly generated starting points), the success ratios (SR) and the CPU times of WalkSAT/GSAT, and in the last column, the adjusted CPU times of DLM-99-SAT. Note that our algorithm has 100% success ratio for all the benchmark instances tested.

Table 2 also lists the results of applying DLM-2000-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 \( par16 \) and \( hanoi \) classes.
Table 3: Comparison of performance of DLM-2000-SAT with LSDL (Choi, Lee, & Stuckey 1998) for solving some hard graph coloring problems. The timing results of LSDL were collected from a SUN Sparc classic model, unknown.

<table>
<thead>
<tr>
<th>Problem ID</th>
<th>Succ. Ratio</th>
<th>CPU Sec.</th>
<th>Num. of Flips</th>
<th>LSDL</th>
<th>GENET</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>g125-17</td>
<td>10/10</td>
<td>41.4</td>
<td>434183</td>
<td>282.0</td>
<td>192.0</td>
<td></td>
</tr>
<tr>
<td>g125-18</td>
<td>10/10</td>
<td>4.8</td>
<td>22018</td>
<td>45</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>g250-15</td>
<td>10/10</td>
<td>17.7</td>
<td>2437</td>
<td>0.418</td>
<td>0.328</td>
<td></td>
</tr>
<tr>
<td>g250-29</td>
<td>10/10</td>
<td>193.1</td>
<td>289962</td>
<td>876.0</td>
<td>678.0</td>
<td></td>
</tr>
</tbody>
</table>

*: Calculated from CPU times reported in minutes (Choi, Lee, & Stuckey 1998).

Table 4: Comparison of performance of DLM-2000-SAT with GRASP (Marques-Silva & Sakalla 1999) on some typical DIMACS benchmarks. The timing results of GRASP were collected from a SUN SPARC 5/85 computer. * stands for 'not solved'.

<table>
<thead>
<tr>
<th>Problem Class</th>
<th>Succ. Ratio</th>
<th>CPU Sec.</th>
<th>Num. of Flips</th>
<th>GRASP Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>f-</td>
<td>10/10</td>
<td>7.7</td>
<td>487198*</td>
<td>-</td>
</tr>
<tr>
<td>g-</td>
<td>10/10</td>
<td>64.3</td>
<td>187150*</td>
<td>-</td>
</tr>
<tr>
<td>pari6</td>
<td>10/10</td>
<td>0.25**</td>
<td>81022*</td>
<td>0.4</td>
</tr>
<tr>
<td>pari16</td>
<td>10/10</td>
<td>108*</td>
<td>1.4 x 10**</td>
<td>9844*</td>
</tr>
<tr>
<td>hanoi</td>
<td>10/10</td>
<td>7777*</td>
<td>8.7 x 10**</td>
<td>14480*</td>
</tr>
</tbody>
</table>

*: Averages computed using values in Table 2
**: Averages computed using values in Table 1

Table 3 compares our algorithm with a typical heuristic-repair method, LSDL (Choi, Lee, & Stuckey 1998), on hard graph coloring problems. Our algorithm outperforms LSDL on g125-17 and g250-29 by using 60% less CPU time. The reason that it uses more time to solve g125-18 and g250-15 is due to the complexity of maintaining a queue of historical points. Hence, for very simple problems, there is no apparent advantage of using our proposed global-search strategy.

Table 4 compares our method to a well-known backtracking and search-pruning algorithm (Marques-Silva & Sakalla 1999). Since this is a complete method that can prove unsatisfiability, it performs much slower than our proposed global-search method.

References


