Jan. 2005, Vol.20, No.1, pp.3–17 *J. Comput. Sci. & Technol.*

Penalty Formulations and Trap-Avoidance Strategies for Solving Hard Satisfiability Problems

Benjamin W. Wah[1] and Zhe Wu[2]

[1]*Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL 61801, U.S.A.*

[2]*Oracle Corporation, One Oracle Drive, Nashua, NH 03062, U.S.A.*

E-mail: wah@uiuc.edu; zhewu@engineering.uiuc.edu

Received November 1, 2004; revised November 30, 2004.

Abstract In this paper we study the solution of SAT problems formulated as discrete decision and discrete constrained optimization problems. Constrained formulations are better than traditional unconstrained formulations because violated constraints may provide additional forces to lead a search towards a satisfiable assignment. We summarize the theory of extended saddle points in penalty formulations for solving discrete constrained optimization problems and the associated discrete penalty method (DPM). We then examine various formulations of the objective function, choices of neighborhood in DPM, strategies for updating penalties, and heuristics for avoiding traps. Experimental evaluations on hard benchmark instances pinpoint that traps contribute significantly to the inefficiency of DPM and force a trajectory to repeatedly visit the same set of or nearby points in the original variable space. To address this issue, we propose and study two trap-avoidance strategies. The first strategy adds extra penalties on unsatisfied clauses inside a trap, leading to very large penalties for unsatisfied clauses that are trapped more often and making these clauses more likely to be satisfied in the future. The second strategy stores information on points visited before, whether inside traps or not, and avoids visiting points that are close to points visited before. It can be implemented by modifying the penalty function in such a way that, if a trajectory gets close to points visited before, an extra penalty will take effect and force the trajectory to a new region. It specializes to the first strategy because traps are special cases of points visited before. Finally, we show experimental results on evaluating benchmarks in the DIMACS and SATLIB archives and compare our results with existing results on GSAT, WalkSAT, LSDL, and Grasp. The results demonstrate that DPM with trap avoidance is robust as well as effective for solving hard SAT problems.

Keywords knowledge representation and reasoning, techniques of algorithms, constraint satisfaction, Boolean satisfiability, penalty formulation, saddle point, search

1 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:

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

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

In this paper, we formulate a SAT problem in two forms:

- Discrete decision problem:

$$U_j(x) = 0, \quad \forall j \in \{1, 2, \ldots, n\}; \tag{2}$$

- Discrete optimization problem:

$$\text{minimize objective subject to } U_j(x) = 0, \quad \forall j \in \{1, 2, \ldots, n\}. \tag{3}$$

Here, $U_j(x)$ is a binary expression equal to zero when the j-th clause is satisfied and to one otherwise, and n is the number of clauses. The selection of a suitable objective in (3) is discussed in Section 4. We convert each formulation into a discrete penalty function and finding satisfiable assignments using efficient global-search strategies.

We have used a constrained formulation instead of the traditional unconstrained formulation because constraints allow a search strategy to better focus its effort on unsatisfied clauses. We have found experimentally that some clauses are more difficult to satisfy than others and need to be handled differently. By choosing a constrained formulation and by assigning dynamically changing penalties according to the duration that the corresponding clause is unsatisfied, our search strategy has a better chance to focus on hard-to-satisfy clauses and find a satisfiable assignment. In contrast, all clauses in an unconstrained formulation have some fixed relative weights. Fixing the weights is undesirable because different clauses may have different degrees of difficulty to be satisfied at different times during a search, making it hard to choose suitable weights ahead of time.

This paper is organized as follows. We survey existing work on solving SAT problems in Section 2 and in-

*Regular Paper

Supported by the National Science Foundation of USA under Grant No.NSF IIS 03-12084.

troduce the extended saddle point condition (ESPC) in discrete space and the associated discrete penalty method (DPM) in Section 3. The theory and formulations discussed here have been extended significantly from the early work of Shang and Wah[1]. We discuss in Section 4 various considerations in choosing objective functions, neighborhoods, updates of penalties, and trap avoidance. We then present a basic implementation of DPM used in [1] for solving SAT problems and explain why some hard instances cannot be solved. To solve those hard instances, we propose in Sections 5 and 6, respectively, efficient trap-avoidance strategies based on a constrained decision formulation and a constrained optimization formulation. The general idea of trap avoidance is to avoid visiting repeatedly the same regions in the penalty-function space. Finally, Section 7 presents our experimental results and compares our algorithms to other well-known methods in this area.

2 Previous Work

Many algorithms and heuristics have been developed for solving SAT problems. In this section, we classify existing methods based on their problem formulations, first according to whether the problem variables are discrete, and second according to whether constraints are included. Methods in each class are further classified according to whether they are *complete* (proving feasibility as well as infeasibility) or *incomplete* (finding a feasible solution if one exists).

2.1 Discrete Formulations

In a discrete formulation, a Boolean variable can be mapped to false (0) or true (1). This mapping is natural for SAT problems.

a) A *discrete constrained decision formulation*, defined in (1), entails the search of solutions that satisfy all the clauses. Existing methods in this class are generally complete methods. Examples include resolution[2−5], backtracking[6], and consistency testing[7]. Due to their exhaustive nature, these methods are expensive to use and normally have difficulty in solving large problems.

There is little work on incomplete methods for solving problems using a constrained decision formulation because incomplete methods generally focus on all the constraints in a single function, rather than individual constraints. However, constraints are useful for providing additional forces to lead a search towards a satisfiable assignment. By combining constraints into a dynamic penalty function, we present in Section 5 new trap-avoidance strategies when looking for extended saddle points.

b) A *discrete unconstrained formulations* involves the minimization of $f(x)$, an objective function on the number of unsatisfied clauses. Obviously, a solution is reached when $f(x)$ is zero.

Methods in this class are generally incomplete methods and cannot prove infeasibility. They can be classified as local- and global-search methods. Since local-search methods may be trapped by local minima in the objective space, various global-search strategies have been proposed. Next, we discuss briefly some existing methods using unconstrained formulations.

Gu[7−11] proposed a number of local search and parallel local search for solving SAT problems. These strategies include iterative perturbation of trajectories and randomized search for overcoming local minima.

Selman *et al.* proposed GSAT[12], a randomized local search that takes the best possible moves whenever possible, and that randomly picks one variable and flips its assignment when there are several moves (or flips) with the same effect. Flat moves, or sideway-moves, are allowed to better explore plateaus in the variable space. GSAT can quickly solve randomly generated 3-SAT problems with up to 2,000 variables.

WalkSAT[13−15] adds random walks ("noise") in GSAT by picking a variable in some unsatisfied clauses with probability p and flips its assignment, and by performing greedy local search with probability $1-p$. It resembles simulated annealing (SA) that accepts descents with a predetermined probability and allows a certain level of ascents in variable space. WalkSAT has been very successful in solving many hard SAT problems. Some theoretical analysis of GSAT/WalkSAT can be found in [16, 17].

Tabu search[18] is a method that records previously-seen patterns using a simple data structure and tries to avoid those patterns in the future. A possible implementation is to maintain a tabu list in order to force a search to explore unknown/unvisited regions in the variable space. Its underlying idea of avoiding visits of historical points is quite general and can be found in many global-search heuristics. In fact, the two trap-avoidance strategies proposed in Sections 5 and 6 can be classified as Tabu search, although we use constrained formulations instead of unconstrained ones. An adaptive way of performing Tabu search can be found in [19].

There are algorithms that address each clause individually by introducing a weight on each[15,20−23] and by updating the weights when descents cannot be performed. These help a search overcome local minima and find solutions quickly. Recently, there are new algorithms on learning conflict clauses for reducing search complexity of SAT problems[24,25].

Stochastic methods, such as GA and SA, have systematic mechanisms to bring a search out of local minima. However, as reported in [14] and based on our experience in applying SA and CSA[26], they are not effective for solving large SAT problems. Their major difficulty lies in their use of exceedingly slow cooling schedules when solving large SAT problems in order for the search to converge to satisfiable assignments. There is some recent research on combining global optimization schemes, like GA, with local-search methods[27].

c) A *discrete constrained optimization formulation* represents a SAT instance as a constrained optimization problem in which each clause is defined as a constraint. In a typical approach, a SAT instance is formulated as an integer linear programming (ILP) problem and solved by existing algorithms, like branch-and-bound[28], cutting-plane[29], and interior-point[30,31] methods. Although these methods sometimes perform better than resolution, they are computationally expensive and cannot solve hard-to-satisfy instances.

A second approach adds an artificial objective function $N(x)$ to the constraints defined on the clauses. The following formulation adds an objective on the number of unsatisfied clauses[1,32]:

$$\min_{x\in\{0,1\}^m} N(x) = \sum_{i=1}^{n} U_i(x) \qquad (4)$$
$$\text{subject to } U_i(x) = 0, \quad \forall i \in \{1, 2, \ldots, n\}.$$

Based on this formulation, the discrete penalty method (DPM) has successfully solved many hard instances in the DIMACS archive[1]. However, this objective does not add any new information because it is simply a summation of all the constraint functions. Hence, its benefit in guidance during a search is doubtful.

2.2 Continuous Formulations

In a continuous formulation, a SAT problem is transformed in such a way that a solution in continuous space will also be a solution to the original problem in discrete space. The approach is flexible because a variable does not have to be committed a binary value prematurely but can take continuous values within a range. However, search algorithms in continuous space are very expensive to apply. Algorithms in this class have been developed for unconstrained and constrained formulations.

a) A *continuous unconstrained formulation* defines an objective as follows:

$$\min_{x\in R^m} f(x) = \sum_{i=1}^{n} C_i(x), \qquad (5)$$

where R is the set of real numbers, and $C_i(x)$ is a function of clause C_i:

$$C_i(x) = \prod_{j=1}^{m} a_{i,j}(x_j) \qquad (6)$$

and

$$a_{i,j}(x_j) = \begin{cases} (1-x_j)^2, & \text{if } x_j \text{ in } C_i; \\ x_j^2, & \text{if } \bar{x}_j \text{ in } C_i; \\ 1, & \text{otherwise.} \end{cases}$$

Many existing methods can be applied to solve (5). Typical local-search methods include gradient descent, conjugate gradient, Quasi-Newton methods, and sequential quadratic programming (SQP). They are fast but may be trapped easily by local minima[8,10,11] and do not work well for large SAT instances with thousands of variables. Global-search techniques, such as clustering, generalized-gradient, Bayesian, stochastic, and trajectory methods[33,34] can also be applied; however, they are usually much more computationally expensive than descent methods.

b) A *continuous constrained optimization formulation* defines a new objective together with the set of constraints in (6):

$$\min_{x\in R^m} f(x) \qquad (7)$$
$$\text{subject to } C_i(x) = 0, \quad \forall i \in \{1, 2, \ldots, n\}.$$

Existing approaches generally construct $f(x)$ as a (redundant) combination of all the constraint functions.

Typical methods for solving continuous constrained optimization problems include Lagrange-multiplier methods[35], SQP[36], CSA[26,37], and genetic algorithms. Our experience with continuous formulations is that they do not reduce the number of local minima, and continuous algorithms are an order-of-magnitude more expensive to apply than the corresponding discrete algorithms[38].

Previous results in the area motivate us to study global-search strategies based on constrained decision/optimization formulations. A SAT instance can be formulated naturally as a constrained decision problems or as a constrained optimization problem, provided that a meaningful objective function can be defined. We discuss in Sections 5 and 6 two such formulations. In the next section, we present the theory of extended saddle points for handling constrained formulations of discrete problems.

3 Theory of Extended Saddle Points

In this section, we summarize briefly the theory of extended saddle points[1,39−41] for solving general discrete constrained optimization problems. The theory was first introduced in [1, 32] for solving SAT problems, although it was incomplete in the sense that it only provided a sufficient, but not necessary, condition for a point to be a constrained local minimum[40,41]. Moreover, only simple global-search strategies were proposed in the original paper[1,32].

Consider a *discrete equality-constrained nonlinear optimization problem*:

$$(P_d) \qquad \min_{\boldsymbol{x}} f(\boldsymbol{x}) \qquad (8)$$
$$\text{subject to } h(\boldsymbol{x}) = 0$$

where $\boldsymbol{x} = (x_1, \ldots, x_n)^{\mathrm{T}} \in \mathcal{D}^n$ is a vector of discrete variables, and $\boldsymbol{h}(\boldsymbol{x}) = (h_1(\boldsymbol{x}), \ldots, h_m(\boldsymbol{x}))^{\mathrm{T}}$ is a vector of constraint functions. In general, both $f(\boldsymbol{x})$ and $\boldsymbol{h}(\boldsymbol{x})$ are nonlinear, and analytic in closed forms or procedural.

The goal of solving P_d is to find a constrained local minimum $\boldsymbol{x}^*$ with respect to $\mathcal{N}_d(\boldsymbol{x}^*)$, the discrete neighborhood of $\boldsymbol{x}^*$. Since the discrete neighborhood of a point is not well defined in the literature, it is up to the user to define the concept. Intuitively, $\mathcal{N}_d(\boldsymbol{x})$ represents points that are perturbed from $\boldsymbol{x}$, with no requirement that there be valid state transitions from $\boldsymbol{x}$.

Definition 1. *Discrete neighborhood* $\mathcal{N}_d(\boldsymbol{x})$[42] *of* $\boldsymbol{x} \in \mathcal{D}^n$ *in discrete space is a* finite *user-defined set of points* $\{\boldsymbol{x}' \in \mathcal{D}^n\}$ *such that* $\boldsymbol{x}'$ *is reachable from* x *in one step, that* $\boldsymbol{x}' \in \mathcal{N}_d(\boldsymbol{x}) \Longleftrightarrow \boldsymbol{x} \in \mathcal{N}_d\boldsymbol{x}'$, *and that it is possible to reach every* $\boldsymbol{x}''$ *from any* $\boldsymbol{x}$ *in one or more steps through neighboring points.*

Definition 2. *Point* $\boldsymbol{x}^*$ *is a* CLM_d, *a constrained local minimum of* P_d *with respect to points in* $\mathcal{N}_d(\boldsymbol{x}^*)$, *if* $\boldsymbol{x}^*$ *is feasible and* $f(\boldsymbol{x}^*) \leqslant f(\boldsymbol{x})$ *for all feasible* $\boldsymbol{x} \in \mathcal{N}_d(\boldsymbol{x}^*)$.

There are two distinct features of CLM_d. First, the set of CLM_d of P_d are neighborhood dependent, and a point may be a CLM_d under one definition of neighborhood but may not be one under another. However, all CLM_d's are guaranteed to be feasible, even in the extreme case in which the neighborhood of each point includes only itself. The fact that CLM_d's are neighborhood dependent is not critical in constrained searches, because our goal is to find feasible solutions that are better than their neighboring points. As long as a consistent neighborhood is used throughout a search, a CLM_d found will be a local minimum with respect to its neighborhood. Second, a discrete neighborhood has a *finite* number of points. Hence, the verification of a point to be a CLM_d can be done by comparing its objective value against that of its *finite* number of neighbors. This feature allows the search of a descent direction in discrete space to be done by enumeration or by greedy search.

Definition 3. *The* ℓ_1*-penalty function for* P_d *is defined for* $\alpha \in \mathcal{R}^m$ *as follows*:

$$L_d(\boldsymbol{x}, \alpha) = f(\boldsymbol{x}) + \alpha^{\mathrm{T}}|\boldsymbol{h}(\boldsymbol{x})|. \tag{9}$$

Theorem 1. *Necessary and sufficient extended saddle-point condition (ESPC) on* CLM_d *of* P_d[1,39–41,43]. *Suppose* $\boldsymbol{x}^* \in \mathcal{D}^w$ *is a point in the discrete search space of* P_d. *Then* $\boldsymbol{x}^*$ *is a* CLM_d *of* P_d *if and only if there exist finite* $\alpha^* \geqslant 0$ *such that the following is satisfied for all* $\boldsymbol{x} \in \mathcal{N}_d(\boldsymbol{x}^*)$ *and* $\alpha \in \mathcal{R}^m$:

$$L_d(\boldsymbol{x}^*, \alpha) \leqslant L_d(\boldsymbol{x}^*, \alpha^{**}) \leqslant L_d(\boldsymbol{x}, \alpha^{**}), \tag{10}$$

where $\alpha^{**} > \alpha^* \geqslant 0$.

We do not show the proof here due to space limitation[40,41].

Intuitively, (10) shows that a local minimum of (9) with respect to $\boldsymbol{x}$ corresponds to a CLM_d of P_d (second inequality of (10)) when α^{**} is larger than some threshold α^* such that all the constraints of P_d are forced to be satisfied (first inequality of (10)). Since a local minimum of (9) can be found easily by many existing search algorithms, a CLM_d of P_d can be found by gradually increasing α^{**}, while minimizing $L_d(\boldsymbol{x}, \alpha^{**})$ until $\alpha^{**} > \alpha^*$. This is described in the following algorithm.

```
$\alpha \to 0$;
repeat
    for $i = 1, \ldots, m$ do;
        if ($h_i(x) \neq 0$ and $\alpha_i < \bar{\alpha}_i$) $\alpha_i \leftarrow \alpha_i + \delta$
    end_for
    repeat
        perform descents of $L_d(x, \alpha)$ w.r.t. $x$;
    until a local minimum of $L_d(x, \alpha)$ is found;
until ($\alpha_i > \bar{\alpha}_i$ for all $h_i(x) \neq 0$) or a $CLM_d$ is
    found
```

Fig.1. Discrete penalty method (DPM), a procedure to look for CLM_d of P_d.

Fig.1 illustrates an implementation the *discrete penalty method* (DPM) for finding CLM_d. Assuming α^{**} has been found in the outer loop and according to the second inequality in (10), the inner loop looks for a local minimum of $L_d(\boldsymbol{x}, \alpha^{**})$ in order to find $\boldsymbol{x}^*$. If a feasible solution to P_d is not found at the local minimum $\boldsymbol{x}$ of $L_d(\boldsymbol{x}, \alpha^{**})$, the penalties corresponding to the violated constraints are increased. The process is repeated until a CLM_d is found or when α^{**} is larger than $\bar{\alpha}$, where $\bar{\alpha}$ is chosen to be so large that it exceeds α^*.

Because $L_d(\boldsymbol{x}, \alpha^{**})$ may have many local minima and some of them do not correspond to CLM_d even when $\alpha^{**} > \alpha^*$, it is possible for the iterative procedure in Fig.1 to terminate without finding a CLM_d. To cope with this issue, we discuss two additional strategies to look for CLM_d.

First, when α^{**} reaches its upper bound during a search but a local minimum of $L_d(\boldsymbol{x}, \alpha^{**})$ does not correspond to a CLM_d of P_d, then a different local minimum of the function will need to be found. Instead of restarting the search from a new starting point, reducing α^{**} will change the terrain and "lower" the barrier of the penalty function, thereby allowing a local search to continue on the same trajectory and move to another local minimum of the penalty function. By repeatedly increasing α^{**} to its upper bound and reducing it to some lower bound, a local search algorithm will be able to visit multiple local minima of the penalty function. Alternatively, it is possible to escape from a local minimum of the penalty function by using a global search algorithm in the inner loop. Since these two strategies offset each other in their effects, only one of them will need to be applied.

Second, the ease of finding a CLM_d depends on the number of CLM_d in the search space of P_d, which in turn depends on the neighborhood function chosen. If the neighborhood of each point is the entire search space itself, then any CLM_d is also a constrained global minimum, and finding a CLM_d amounts to global optimization. On the other hand, if the neighborhood of each point is only the point itself, then any feasible point in the search space is a CLM_d. In this case, since the neighborhood is limited, only random probing can be applied, and finding a CLM_d amounts to feasibility search. In practice, we choose the neighborhood of each point to two methods involve steps that are carried out periodically and not specific to the case when a trajectory is inside traps.

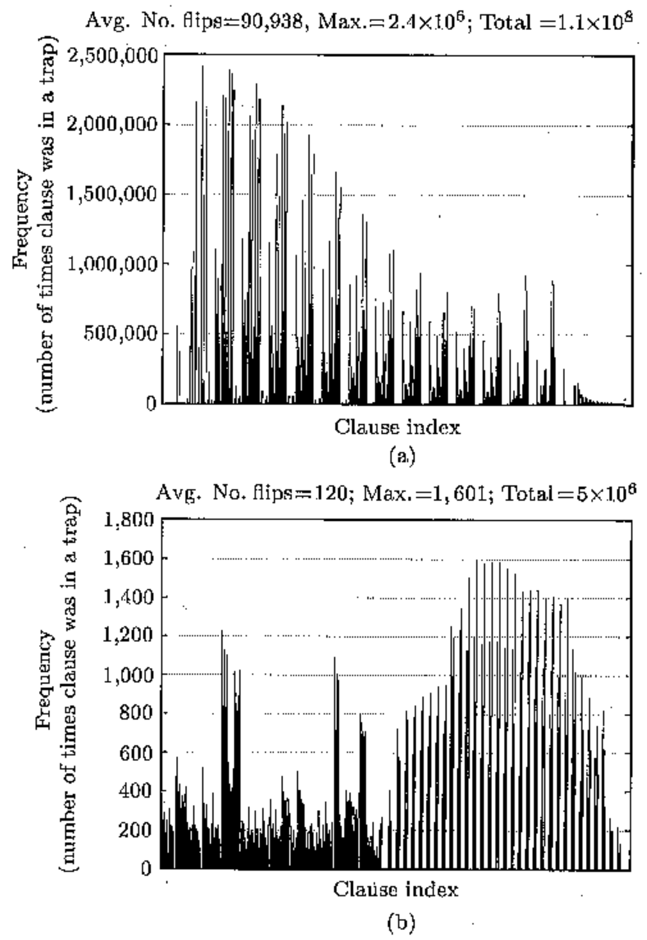

Fig.2. Large disparity between the maximum and the average numbers of times a clause is in traps. (a) *Hanoi*4. (b) *Par*16-1.

a) *Flipping multiple variables at a time.* Since a trap is defined with respect to the perturbation of one variable, we can perturb multiple variables at a time to see if L_d decreases. This is impractical due to the large number of neighborhood points that need to be considered.

b) *Random restarts.* Although restarts are effective to bring a trajectory out of traps, valuable historical information accumulated will be lost. It is also impractical to maintain historical information on all the traps visited in a trajectory.

c) *Extra penalties on clauses in traps.* The idea is to add extra penalties on unsatisfied clauses inside traps, leading to very large penalties for unsatisfied clauses that are trapped more often and making these clauses less likely to be unsatisfied in the future. This is an *indirect* strategy because it only imposes conditions to make it less likely for a trajectory to repeat the same traps, rather than identifying traps explicitly and pushing a trajectory away from them. The strategy has been found to be very useful and is evaluated in Section 5.

d) *Keeping historical information on trajectories.* The idea is to avoid visiting points, whether inside traps or not, close to points that have already been visited before in a trajectory. It can be implemented by modifying the penalty function in such a way that, if a trajectory gets close to points visited before, then an extra penalty will take effect and force the trajectory to a new region. Of course, one can only keep track of a part of a search trajectory due to space and computation limits. Experimental results show that even keeping a small part of the historical information is very helpful in reducing the time to find satisfiable assignments. This is a *direct* strategy because it stores explicit information on points visited before. It is investigated thoroughly in Section 6.

e) *Periodic decreases of penalties* can bring a trajectory to a new region in a penalty-function space. Hence, if a trajectory is inside a trap, it will be out of the trap once the space changes. This *indirect* trap-avoidance strategy has been studied in the basic DPM in [1].

4.3 Basic DPM for SAT

Fig.3 shows a basic implementation of DPM[1] that uses some of the concepts discussed in this section: constant $f(x)$, $\mathcal{N}_2(x)$, and periodic reduction of all penalties by a common factor. In addition, it uses heuristics based on tabu lists[18] and flat moves[14]. We explain each step of this algorithm when we present our trap-avoidance strategy in the next section.

Table 1 lists the average performance of our current implementation of *DPM-BASIC-SAT* in solving DIMACS/SATLIB benchmark problems, each evaluated from ten randomly generated starting points.

Although quite simple, *DPM-BASIC-SAT* can find solutions within seconds to most satisfiable DIMACS benchmarks, such as all the problems in the *aim*, *ii*, *jnh*, *par*8, and *ssa* classes, and most problems in SATLIB, like uniform 3-SAT *uf*, flat graph coloring *flat*, and morphed graph coloring *sw*. *DPM-BASIC-SAT* is either faster than, or at least comparable to, competing algorithms like GSAT and Grasp[45]. However, it has difficulty in solving problems in the *par*16, *hanoi*, *g*, *f*2000, and *par*32 classes.

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

Fig.3. *DPM-BASIC-SAT*[1]: An implementation of the basic discrete penalty method in Fig.1 for solving SAT.

be rich enough in order to achieve a balance between the number of neighbors of each point and the number CLM_d in the search space.

4 Solving SAT Using Penalty Formulations

In this section, we describe the solution of SAT as a discrete penalty search to find extended saddle points. The penalty search, when augmented by new heuristics presented in the following sections, provides a powerful tool to solve hard-to-satisfy SAT instances.

4.1 Objective Formulations

Since the original discrete decision problem (2) does not have an objective $f(\boldsymbol{x})$, an artificial objective needs to be created before the problem can be transformed into a discrete penalty function (9). We have studied the following alternative objective functions.

a) *Constant* $f(\boldsymbol{x})$. Variations include setting $f(\boldsymbol{x})$ to zero and to the number of violated constraints[1]. In this case, the objective does not have any effect on search in a penalty formulation. Although results in [1] demonstrate good performance using this formulation, better performance can be accomplished by using a suitably chosen objective. Section 5 presents global-search strategies based on setting $f(\boldsymbol{x})$ to zero.

b) $f(\boldsymbol{x}) = -\sum_i v_i(\boldsymbol{x})$, where $v_i(\boldsymbol{x})$ is the number of variables in Clause C_i that can make C_i satisfied. For example, $v_1(\boldsymbol{x}) = 1$ when $C_1 = x_1 \vee x_2$ and $\boldsymbol{x} = (x_1, x_2) = (1, 0)$, because only x_1 can make this clause satisfied. Note that $v_i(\boldsymbol{x}) = 0$ when C_i is unsatisfied. The intuitive reason for choosing this objective is to maximize the number of variables that can make at least one clause satisfied, since a trajectory will be quite close to a solution when there are many variables that can satisfy different clauses. However, this choice does not perform well experimentally when applied to solve hard SAT benchmarks. A possible reason is that there is no direct connection between such $f(\boldsymbol{x})$ and constraint satisfaction, and a small $f(\boldsymbol{x})$ does not necessarily mean that it is better in terms of constraint satisfaction.

c) *Setting* $f(\boldsymbol{x})$ *to be the sum of a few hard-to-satisfy constraints.* This approach aims to emphasize a few constraints in the objective. It does not work well because whether a constraint is hard to satisfy may depend on its current assignment, and it is hard to identify all of them ahead of time.

d) $f(\boldsymbol{x}) = \sum_i T(u_i(\boldsymbol{x}))$, where $u_i(\boldsymbol{x})$ is the number of times that Clause C_i is unsatisfied, and T is an exponential function. The goal here is penalize clauses that are unsatisfied more often than others by assigning a large weight to them in order to force them into satisfaction in the future. This approach does not work well because the constraint part in the discrete penalty function will carry smaller and smaller relative weights as more constraints are satisfied, making the search biased too much towards its historical information.

e) *Setting* $f(\boldsymbol{x})$ *to measure the coverage of variable space by a search trajectory.* This approach finds a trajectory that covers the variable space evenly, while avoiding the repetition of the same subspaces searched before. It is intuitively sound because it aims to explore the variable space efficiently, although its success depends on a suitable coverage measure that can be implemented efficiently. We study in Section 6 *distance_penalty*$(\boldsymbol{x})$, a new objective that measures the sum of Hamming distances between the current point and points visited recently in the trajectory.

In short, our experimental results show that the first and the last objective functions work well in solving SAT problems. There are other choices that may be intuitively sound but do not lead to constraint satisfaction.

4.2 Discrete Penalty Method

There are three components in DPM that may affect performance:

A) *Choice of neighborhood.* The neighborhood of a point defines the set of points with different original-variable assignments that need to be evaluated when carrying out a greedy search (or hill climbing) in the penalty-function space. Its choice involves trade-offs between the quality of descent directions and the overhead in finding them.

The main advantage of using small neighborhoods is its low overhead in finding descent directions. However, small neighborhoods with only nearby points may not be as good as large neighborhoods with a combination of nearby and distant points because the latter may lead to better descent directions and have a smaller chance of getting stuck in a trap (defined later) or in a flat region.

In this paper, we have studied four different neighborhoods $\mathcal{N}_d(\boldsymbol{x})$ of $\boldsymbol{x}$:

a) $\mathcal{N}_1(\boldsymbol{x})$ is the set of variable assignments that differ from $\boldsymbol{x}$ by one variable. That is, $\boldsymbol{x}$ can be changed to $\boldsymbol{x}' \in \mathcal{N}_1(\boldsymbol{x})$ by flipping just one variable, and vice versa.

b) $\mathcal{N}_2(x)$ is the set of variable assignments that differ from $\boldsymbol{x}$ by one variable, and the flip must be a variable in an unsatisfied clause. That is, $\mathcal{N}_2(\boldsymbol{x})$ only allows flips in $\mathcal{N}_1(\boldsymbol{x})$ that will change at least one unsatisfied clause into satisfaction.

c) $\mathcal{N}_3(\boldsymbol{x})$ is the set of variable assignments that differ from x by multiple variables. That is, $\boldsymbol{x}$ will change to $\boldsymbol{x}' \in \mathcal{N}_3(\boldsymbol{x})$ by flipping one or more variables, and vice versa. Normally, we set an upper bound on the number of flips that can be performed all at once.

d) $\mathcal{N}_4(\boldsymbol{x})$ is the set of variable assignments that differ from $\boldsymbol{x}$ by multiple variables, and each of those flips must be a variable in an unsatisfied clause. This is similar to $\mathcal{N}_2(\boldsymbol{x})$ except that it allows multiple flips at the same time.

Obviously, $\mathcal{N}_2(\boldsymbol{x})$ is smaller in size than $\mathcal{N}_1(\boldsymbol{x})$, and $\mathcal{N}_4(\boldsymbol{x})$ is smaller than $\mathcal{N}_3(\boldsymbol{x})$. Both $\mathcal{N}_3(\boldsymbol{x})$ and $\mathcal{N}_4(\boldsymbol{x})$ allow multiple flips at the same time, hence providing more effective exploration of a variable space but at a significant overhead.

Our experimental results show that $\mathcal{N}_2(\boldsymbol{x})$ is better than $\mathcal{N}_1(\boldsymbol{x})$ because the latter is more closely related to unsatisfied clauses and is more pertinent to our goal of constraint satisfaction. Similarly, $\mathcal{N}_3(\boldsymbol{x})$ is better than

$\mathcal{N}_4(\boldsymbol{x})$ for the same reason. Both $\mathcal{N}_3(\boldsymbol{x})$ and $\mathcal{N}_4(\boldsymbol{x})$ are much more expensive to apply in a neighborhood search than $\mathcal{N}_2(\boldsymbol{x})$, with only a marginal gain in terms of the quality of descent directions found. For this reason, we select $\mathcal{N}_2(\boldsymbol{x})$ in our algorithms. Note that our implementation of neighborhood search is similar to that in GSAT because they both carry out greedy local searches.

Based on $\mathcal{N}_2(\boldsymbol{x})$, we define a *trap* as a combination of $\boldsymbol{x}$ and α such that a point in it has one or more unsatisfied clauses, and any change to a *single* variable in $\boldsymbol{x}$ will cause $L_d(\boldsymbol{x}, \alpha)$ to increase. Note that a satisfiable assignment is not a trap because all its clauses are satisfied, even though $L_d(\boldsymbol{x}, \alpha)$ may increase when $\boldsymbol{x}$ is perturbed.

B) *Strategies for updating penalties.* As is discussed earlier, updates of penalties are critical in bringing a trajectory out of traps in the original variable space.

We have tried to increase the penalties of unsatisfied clauses periodically and when trajectories get stuck in flat regions or in traps. Experimentally, we have found that periodic increases is not beneficial and may actually lead to oscillations in a trajectory. The reason is that a search may not have enough time to explore a region in detail before it is forced to leave, after increasing its penalties. As a result, we increase penalties only when a search reaches a trap or a flat region.

As is discussed in Section 3, it is necessary to reduce α in DPM periodically in order to avoid it growing unbounded when some constraints remain unsatisfied. Since large penalties result in a rugged penalty-function space and make it difficult for a trajectory to escape from infeasible local minima, the penalties need to be decreased periodically in order to change the relative weights of clauses. So far, we have tried the following alternatives.

a) *Setting all penalties to zero periodically.* This strategy does not work well because all historical information accumulated in a trajectory will be lost after resets.

b) *Controlling the ratio of the maximum to the average values of penalties.* The idea here is to limit the "ruggedness" of a penalty function by scaling all penalties when the ratio is larger than a threshold. It does not work well because it is hard to choose a suitable problem-dependent threshold.

c) *Subtracting a common factor from all penalties periodically.* This strategy generalizes well to various SAT instances. We have adopted this strategy in our current implementation.

d) *Subtracting a common factor from all penalties of satisfied clauses periodically.* Intuitively, decreasing the penalties of all satisfied clauses actually increases the relative weights of unsatisfied clauses, thereby making them easier to be satisfied. Unfortunately, this strategy does not prove to be useful experimentally.

e) *Scaling all penalties by a common factor if their average is larger than a threshold.* This strategy is similar to the last two strategies except that it is triggered when the average is larger than a threshold, rather than periodically. It is used in our implementation to solve the *Hanoi* problems but does not generalize well to other problems.

In short, a simple strategy to reduce all penalties periodically works well. More complex strategies do not because they have problem-dependent and difficult-to-tune parameters.

C) *Strategies for trap avoidance.* The occurrence of traps is a consequence of the interleaved application of descents in the original-variable space and ascents in the penalty space, and is specific to our implementation of Fig.1. Due to the use of neighborhood function $\mathcal{N}_2(\boldsymbol{x})$, we flip one variable at a time in our implementation if the flip can decrease L_d and continue until no new single-variable flips can be found.

Although traps cannot be prevented in general, it is undesirable to get into the same set of traps repeatedly because it is obviously wasteful. To this end, we study trap-avoidance strategies using indirect controls in Section 5 and direct controls in Section 6.

A typical scenario is as follows. A clause is initially unsatisfied but becomes satisfied after a few flips due to increases of its α. It then becomes unsatisfied again after a few more flips due to increases of α of other unsatisfied clauses. Such cyclic state changes on a set of clauses are tremendously inefficient because the trajectory remains in an unsatisfiable state. Note that, although DPM performs global search in the variable space and can escape from traps after getting there, they cannot prevent a trajectory from revisiting the same traps in the future.

The occurrence of traps is illustrated in a simple implementation of Fig.1. The output profiles show that some clauses are flipped frequently from being satisfied to unsatisfied, meaning that the trajectory traverses in cycles in a small region or is stuck in the original-variable space for an indefinite period of time. To demonstrate 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.

Fig.2 shows that some clauses reside in traps more often than average in using DPM to solve *hanoi4* and *par*16-1, two hard SAT instances in the DIMACS archive. This behavior leads to an inefficient search because the trajectory may be trapped in a small region for a long time.

Ideally, we like a trajectory to never visit the same point twice in solving an optimization problem. This can be achieved if we can store all the points traversed by a trajectory in the past. It is, however, impractical in terms of memory usage and computation overhead in calculating descent directions for the ever-increasing number of points visited before.

We present in the following five approximate methods to force a trajectory not to repeat itself. The first three methods identify explicitly when a trajectory is in traps before taking corrective actions. The remaining

Table 1. Performance of *DPM-BASIC-SAT* in Solving DIMACS/SATLIB SAT Problems. All experiments were run on a 500MHz Pentinum-III computer with Solaris 7. (*aim* is on artificially generated random-3-SAT; *ii* is from inductive inference; *jnh* is on random SAT with variable-length clauses; *par8* is for learning parity functions; *ssa* is on circuit fault analysis; *ais* is on all-interval series; *uf* is on uniform random-3-SAT; *flat* is on "flat" graph coloring; *logistics* is on logistics planning; and *sw* is on "morphed" graph coloring[44].)

Problem ID	Succ. ratio	CPU (s)	No. of flips	Problem ID	Succ. ratio	CPU (s)	No. of flips
aim-50-1-6-yes1-1	10/10	0.01	3,645	aim-50-1-6-yes1-2	10/10	0.01	1,466
aim-50-1-6-yes1-3	10/10	0.01	984	aim-50-1-6-yes1-4	10/10	0.01	2,060
aim-50-2-0-yes1-1	10/10	0.00	987	aim-50-2-0-yes1-2	10/10	0.01	1,169
aim-50-2-0-yes1-3	10/10	0.01	2,736	aim-50-2-0-yes1-4	10/10	0.01	2,292
aim-50-3-4-yes1-1	10/10	0.01	1,639	aim-50-3-4-yes1-2	10/10	0.01	1,126
aim-50-3-4-yes1-3	10/10	0.01	729	aim-50-3-4-yes1-4	10/10	0.01	665
aim-50-6-0-yes1-1	10/10	0.01	199	aim-50-6-0-yes1-2	10/10	0.01	197
aim-50-6-0-yes1-3	10/10	0.00	171	aim-50-6-0-yes1-4	10/10	0.00	148
aim-100-1-6-yes1-1	10/10	0.02	6,031	aim-100-1-6-yes1-2	10/10	0.02	4,512
aim-100-1-6-yes1-3	10/10	0.02	5,768	aim-100-1-6-yes1-4	10/10	0.01	2,912
aim-100-2-0-yes1-1	10/10	0.03	9,460	aim-100-2-0-yes1-2	10/10	0.03	9,473
aim-100-2-0-yes1-3	10/10	0.02	5,077	aim-100-2-0-yes1-4	10/10	0.02	7,797
aim-100-3-4-yes1-1	10/10	0.05	10,503	aim-100-3-4-yes1-2	10/10	0.02	2,783
aim-100-3-4-yes1-3	10/10	0.04	7,667	aim-100-3-4-yes1-4	10/10	0.02	4,898
aim-100-6-0-yes1-1	10/10	0.01	476	aim-100-6-0-yes1-3	10/10	0.01	680
aim-100-6-0-yes1-2	10/10	0.01	229	aim-100-6-0-yes1-4	10/10	0.01	819
aim-200-1-6-yes1-1	10/10	0.17	80,877	aim-200-1-6-yes1-2	10/10	0.07	29,595
aim-200-1-6-yes1-3	10/10	0.15	68,990	aim-200-1-6-yes1-4	10/10	0.06	29,865
aim-200-2-0-yes1-1	10/10	0.44	174,356	aim-200-2-0-yes1-2	10/10	0.15	57,462
aim-200-2-0-yes1-3	10/10	0.10	36,183	aim-200-2-0-yes1-4	10/10	0.33	129,955
aim-200-3-4-yes1-1	10/10	0.54	99,393	aim-200-3-4-yes1-2	10/10	0.11	18,354
aim-200-3-4-yes1-3	10/10	0.11	19,583	aim-200-3-4-yes1-4	10/10	0.53	98,180
aim-200-6-0-yes1-1	10/10	0.02	894	aim-200-6-0-yes1-2	10/10	0.04	1,961
aim-200-6-0-yes1-3	10/10	0.03	1,700	aim-200-6-0-yes1-4	10/10	0.02	632
ii8a1	10/10	0.01	59	ii8a2	10/10	0.00	147
ii8a3	10/10	0.02	365	ii8a4	10/10	0.02	1,068
ii8b1	10/10	0.01	78	ii8b2	10/10	0.01	429
ii8b3	10/10	0.03	772	ii8b4	10/10	0.03	796
ii8c1	10/10	0.01	433	ii8c2	10/10	0.04	1,714
ii8d1	10/10	0.02	1,098	ii8d2	10/10	0.05	2,720
ii8e1	10/10	0.01	307	ii8e2	10/10	0.03	1,767
ii16a1	10/10	0.20	9,360	ii16a2	10/10	0.22	10,116
ii16b1	10/10	0.32	6,673	ii16b2	10/10	0.23	5,395
ii16c1	10/10	0.18	2,696	ii16c2	10/10	0.25	8,303
ii16d1	10/10	0.22	12,289	ii16d2	10/10	0.49	18,479
ii16e1	10/10	0.14	1,042	ii16e2	10/10	0.28	8,132
ii32a1	10/10	0.13	5,478	ii32b1	10/10	0.01	451
ii32b2	10/10	0.03	870	ii32b3	10/10	0.05	2,139
ii32b4	10/10	0.12	6,268	ii32c1	10/10	0.01	254
ii32c2	10/10	0.02	325	ii32c3	10/10	0.04	1,490
ii32c4	10/10	0.41	7,506	ii32d1	10/10	0.03	971
ii32d2	10/10	0.08	2,904	ii32d3	10/10	0.33	8,676
ii32e1	10/10	0.01	168	ii32e2	10/10	0.03	892
ii32e3	10/10	0.06	2,426	ii32e4	10/10	0.04	2,083
ii32e5	10/10	0.11	5,083				
jnh1	10/10	0.01	899	jnh7	10/10	0.01	632
jnh12	10/10	0.02	1,491	jnh17	10/10	0.02	1,250
jnh201	10/10	0.01	155	jnh204	10/10	0.02	2,401
jnh205	10/10	0.03	2,732	jnh207	10/10	0.05	6,829
jnh209	10/10	0.04	4,146	jnh210	10/10	0.01	506
jnh212	10/10	0.24	33,197	jnh213	10/10	0.02	1,459
jnh217	10/10	0.01	381	jnh218	10/10	0.02	1,104
jnh220	10/10	0.08	9,918	jnh301	10/10	0.10	11,039
par8-1-c	10/10	0.03	7,698	par8-2-c	10/10	0.05	14,421
par8-3-c	10/10	0.83	271,275	par8-4-c	10/10	0.07	21,763
par8-5-c	10/10	0.09	26,736	par8-1	10/10	0.13	41,810
par8-2	10/10	0.17	57,521	par8-3	10/10	0.38	122,311
par8-4	10/10	0.15	48,256	par8-5	10/10	0.40	135,212
ssa7552-038	10/10	0.13	16,250	ssa7552-158	10/10	0.07	8,816
ssa7552-159	10/10	0.08	8,084	ssa7552-160	10/10	0.10	13,742
ais6	10/10	0.01	416	ais8	10/10	0.07	7,242
ais10	10/10	0.23	18,916	ais12	10/10	2.19	140,294
uf200-01	10/10	0.14	11,810	uf200-02	10/10	0.20	22,446
uf200-03	10/10	0.07	1,851	uf200-04	10/10	0.11	8,248
uf200-05	10/10	0.17	16,162	uf200-06	10/10	0.66	80,804
uf200-07	10/10	0.13	12,457	uf200-08	10/10	0.08	7,199
uf200-09	10/10	0.17	15,005	uf200-0100	10/10	0.12	12,732
flat100-1	10/10	0.36	108,069	flat100-2	10/10	0.17	49,512
flat100-3	10/10	0.05	11,072	flat100-4	10/10	0.47	150,496
flat100-5	10/10	0.09	23,146	flat100-6	10/10	0.17	46,834
flat100-7	10/10	2.64	859,110	flat100-8	10/10	2.77	900,221
flat100-9	10/10	0.06	16,428	flat100-100	10/10	0.07	18,502
logistics-a	10/10	0.16	17,427	logistics-b	10/10	0.16	18,965
logistics-c	10/10	0.21	16,870	logistics-d	10/10	1.65	48,603

continued on next page

continued from previous page

Problem ID	Succ. ratio	CPU (s)	No. of flips	Problem ID	Succ. ratio	CPU (s)	No. of flips
sw100-1	10/10	0.62	117,577	sw100-2	10/10	1.43	288,571
sw100-3	10/10	0.97	192,017	sw100-4	10/10	1.02	203,461
sw100-5	10/10	0.65	127,605	sw100-6	10/10	1.75	352,747
sw100-7	10/10	0.83	160,884	sw100-8	10/10	0.99	197,997
sw100-9	10/10	0.89	171,101	sw100-10	10/10	0.55	104,940
sw100-11	10/10	0.83	163,650	sw100-12	10/10	1.64	331,097
sw100-13	10/10	0.68	130,571	sw100-14	10/10	0.50	93,632
sw100-15	10/10	0.52	98,793	sw100-16	10/10	1.79	364,416
sw100-17	10/10	0.71	136,542	sw100-18	10/10	0.83	162,783
sw100-91	10/10	1.57	312,889	sw100-92	10/10	1.04	207,944
sw100-93	10/10	0.89	172,169	sw100-94	10/10	1.19	237,928
sw100-95	10/10	0.45	84,771	sw100-96	10/10	0.71	136,622
sw100-97	10/10	0.77	150,486	sw100-98	10/10	1.46	295,163
sw100-99	10/10	1.28	247,702	sw100-100	10/10	0.47	89,026
sw100-8-p0-c5	10/10	1.00	191,275				

5 Trap Avoidance

The penalty function when formulating SAT as a constrained decision problem (2) is:

$$L_d(x, \alpha) = \sum_{i=1}^{n} \alpha_i U_i(x), \tag{11}$$

where $U_i(x)$ is a binary function equal to zero when the i-th clause is satisfied and to one otherwise. Here, we assume that a constant objective function is used.

```
procedure DPM-Trap-Avoidance-SAT
/* An implementation of DPM for solving SAT */
1.  Reduce original SAT problem;
2.  Generate a random starting point using a fixed seed;
3.  Initialize $\alpha_i \leftarrow 0$ and $t_i \leftarrow 0$;
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 $u$, $t_u \leftarrow t_u + \delta_w$
          end_if
8.      Maintain TabuList;
9.      Flip $x_j$;
10.     if $\#_{FlatMoves} > \theta_1$ then
11.         $\alpha_i \leftarrow \alpha_i + \delta_o$;
12.         if $\#_{Adjust} \% \theta_2 = 0$ then
13.             call DECREASE-ALPHA end_if;
14.         call SPECIAL-INCREASE;
15.     end_if
16. end_while
end
procedure SPECIAL-INCREASE
/* Special increases of $\alpha$ on certain clauses when their
   weights are unbalanced */
17. Pick a set of clauses $S$;
18. if $\frac{\max_{i \in S} t_i}{\sum_{i \in S} t_i / n} \geqslant \theta_3$ then
19.     For clause $i$ in $S$ with the largest $t_i$, $\alpha_i \leftarrow \alpha_i + \delta_s$;
20. end_if
end
procedure DECREASE-ALPHA
/* Two alternatives to decrease penalties */
21. Alternative-1: $\alpha_i \leftarrow \alpha_i - \delta_d$ for all clauses
22. Alternative-2: $\alpha_i \leftarrow \alpha_i / \delta_d$ for all clauses when the
        average $\alpha_i$ of all clauses is larger than $\theta_4$
end
```

Fig.4. Pseudo code of *DPM-Trap-Avoidance-SAT*.

Fig.4 shows the pseudo code of DPM with Strategy (c) on trap avoidance (discussed in Section 4) in which extra penalties are added to clauses when they are found in traps[46]. It defines a weight for each penalty, identifies the location of traps, and increases the weights of all unsatisfied clauses each time the search reaches a trap. When an undesirable imbalance happens where some clauses are trapped more often than others, the penalties of clauses with the largest weight are increased in order to force these clauses into satisfaction. In our implementation, an imbalance happens when the ratio of the largest weight to the average is larger than a predefined threshold. If the increments to the corresponding penalties of those trapped clauses are large enough, these clauses are likely to be satisfied in the future. Also, the chance for the trajectory to hit the same traps again is much lower.

We explain each line of the procedures in Fig.4 in detail next.

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 the clause satisfied and propagate the assignment. For example, if a clause has just one variable x_4, then x_4 must be true. 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 t_i (temporary weight) and α_i for Clause i to zero in order to make the experiments repeatable. Note that increases to α_i are faster if t_i 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 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. Such a flip is called a *flat move*[14]. We allow flat moves in order to help a trajectory explore flat regions.

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*[18] that is important in helping a search explore flat regions effectively. Each time a variable is flipped, it will be put in *TabuList* in an FIFO order, and the oldest element will be removed from *TabuList*.

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 moves.

Lines 10–11 increase the penalties of 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 δ in Fig.1. After increasing the penalties of unsatisfied clauses, we increase Counter $\#_{Adjust}$ by one.

Lines 12–13 implement Strategy (e) on trap avoidance (discussed in Section 4) that reduces the penalties of all clauses by calling Procedure *DECREASE-ALPHA* when $\#_{Adjust}$ reaches threshold θ_2 (12 for f, 46 for *par*16, 56 for *par*32, 6 for g, and 40 for *hanoi*4). This step helps change the relative weights of all clauses and allows the trajectory to go to another region in the penalty-function space after the reduction.

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

Line 17 picks S, a problem-dependent set of clauses (for *par*16-1 to *par*16-5, the set of all currently unsatisfied clauses; for others, the set of all clauses).

Lines 18–19 compute the ratio between the maximum and the average weights to see if the ratio is unbalanced, 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 penalty of the clause with the largest weight by δ_s (1 for all problems).

Lines 21–22 provide two alternatives to reducing penalties. The first alternative reduces penalties by a common integer δ_d (set to 1 in our experiments), whereas the second calculates the average of all penalties before decreasing them by a common factor δ_d (= 2) if the average is larger than θ_4 (set to 4.0 in our experiments). The second alternative was used in *DPM-Trap-Avoidance-SAT* when solving *hanoi4* and *hanoi4-simple* because it allowed satisfiable assignments to be found in time shorter by an order of magnitude when compared to using the first alternative[46].

Intuitively, increasing the penalties of unsatisfied clauses in traps can reduce their chance to be in traps again. Fig.5 illustrates this point by plotting the number of times that clauses appear in traps after using *SPECIAL-INCREASE*. When compared to Fig.2, we see that *SPECIAL-INCREASE* has controlled the large imbalance in the number of times that clauses are unsatisfied. For *hanoi*4 (resp. *par*16-1), 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 imbalance is controlled by θ_3 and δ_s. If we use a smaller θ_3 and a larger δ_s, then a better balance

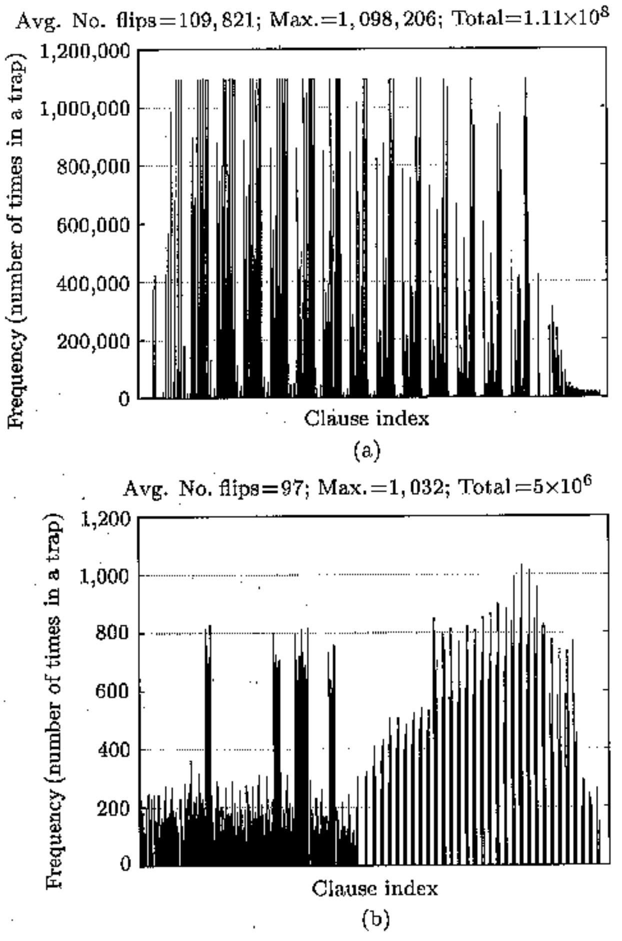

Fig.5. Reduced disparity between the maximum and the average numbers of times a clause is in traps using *SPECIAL-INCREASE*. (a) *Hanoi*4. (b) *Par*16-1.

Table 2. Performance of *DPM-Trap-Avoidance-SAT* in Solving Some Hard SAT Instances and the g-Class Problems That Were not Solved Well Before[1] (Experiments were run on a 500MHz Pentinum III computer with Solaris 7.)

Problem ID	Succ. ratio	CPU (s)	#. flips	Problem ID	Succ. ratio	CPU (s)	#. flips
par16-1	10/10	96.5	1.6×10^7	par16-1-c	10/10	28.8	4,850,828
par16-2	10/10	95.7	1.7×10^7	par16-2-c	10/10	61.0	10,138,948
par16-3	10/10	125.7	2.2×10^7	par16-3-c	10/10	35.3	5,920,445
par16-4	10/10	54.5	9.2×10^6	par16-4-c	10/10	46.1	7,786,958
par16-5	10/10	178.5	3.0×10^7	par16-5-c	10/10	44.6	7,386,779
hanoi4	10/10	14,744	5.4×10^8	hanoi4-simple	10/10	14,236	9.2×10^8
par32-1-c	1/20	8,622	8.9×10^8	g125-17	10/10	144.8	754,560
par32-2-c	1/20	102,590	9.2×10^9	g125-18	10/10	3.98	6,819
par32-3-c	1/20	154,607	1.4×10^{10}	g250-15	10/10	12.9	2,426
par32-4-c	1/20	115,963	1.1×10^{10}	g250-29	10/10	331.4	354,453
f600	10/10	0.664	39,935	f1000	10/10	3.7	217,062
f2000	10/10	16.2	655,100				

can be achieved. However, a better balance does not always lead to better solutions because a search may leave a trap quickly, thereby missing some solutions for hard problems.

Table 2 lists the experimental results on all the hard instances solved by *DPM-Trap-Avoidance-SAT*. We leave comparisons to other algorithms in the area in Section 7. Note that *DPM-Trap-Avoidance-SAT* is one of the few methods that can solve *par*16-1 to *par*16-5, *hanoi*4, *hanoi4-simple*, and *par*32-1-*c* to *par*32-4-*c*, among existing local- and global-search methods for solving SAT.

6 Trap Avoidance

The trap-avoidance strategy presented in the last section is an *indirect* strategy because it only imposes conditions in such a way that makes it less likely for a trajectory to repeat the same traps. In this section we present a *direct* strategy that remembers points visited recently in a trajectory and avoids revisiting them explicitly by imposing penalties on these points. The strategy specializes to trap avoidance because traps are special cases of points visited before. Since these penalties are to be minimized, we have included them in the objective of a constrained optimization problem, leading to the following discrete penalty function with a new heuristic objective:

$$L_d(x,\alpha) = -distance\_penalty(x) + \sum_{i=1}^{n} \alpha_i U_i(x), \qquad (12)$$

where $distance\_penalty(x)$ is the sum of Hamming distances from the current point x in a trajectory to points visited in the recent past. Hence, if a trajectory is stuck in a trap, then the penalties from the current point to points in the trap will be large, leading to a large objective in (12) and steering the trajectory away from the trap. In contrast, if the trajectory is not inside a trap, then the penalties to points visited before will be large, again steering the trajectory away from regions visited before.

The form of $distance\_penalty(x)$ is:

$$distance\_penalty(x) = \Big(-C_L \prod_j (1 - U_j(x)) \Big) + \sum_i \min(\theta_t, |x - x_i^s|), \qquad (13)$$

where θ_t is a positive threshold, $|x - x_i^s|$ is the Hamming distance between x and x_i^s visited before in a trajectory, i is an index to all the points saved, j is an index to all clauses, and C_L is a large positive constant. If x is not a solution, then $(-C_L \cdot \prod_j (1 - U_j(x)))$ in (13) will be zero, and $distance\_penalty(x)$ only measures the sum of Hamming distances. On the other hand, if x is a feasible solution that satisfies all the constraints, then $(-C_L \cdot \prod_j (1 - U_j(x)))$ will be $-C_L$, making the solution point a CLM_d as compared to neighboring feasible points.

Parameter θ_t is used to control the search and put an upper bound on *distance_penalty* so that it will not be a dominant factor in the new penalty function. Without θ_t, the penalty search will prefer a far-away point than a point with less constraint violation, which is not desirable. In our experiments, we set θ_t to be 2. This means that, when the Hamming distance between the current point and each stored historical point of a trajectory is larger than 2, the impact of all the stored historical points on *distance_penalty* will be the same.

Due to limitations in memory usage and computation overhead in calculating descent directions on L_d, we only keep a fixed-size queue of size q_s of historical points and periodically update this queue in an FIFO manner. The period of update is based on w_s flips; namely, after w_s flips, we save the current search point in the queue and remove the oldest element from the queue.

```
procedure DPM-Distance-Penalty-SAT
1. Reduce the original SAT instance;
2. Generate a random starting point using a fixed seed;
3. Initialize $\alpha_i \leftarrow 0$;
4. while solution not found and time not used up do
5.       Pick $x_j \notin TabuList$ that reduces $L_d$ the most;
6.       Flip $x_j$;
7.       If $\#_{Flips} \% w_s = 0$ then
8.           Update the queue on historical points
         end_if
9.       Maintain TabuList;
10.      if $\#_{FlatMoves} > \theta_1$ then
11.          $\alpha_i \leftarrow \alpha_i + \delta_o$;
12.          if $\#_{Adjust} \% \theta_2 = 0$ then
13.              $\alpha_i \leftarrow \alpha_i - \delta_d$; end_if;
14.          end_if
15. end_while
end
```

Fig.6. Pseudo code of *DPM-Distance-Penalty-SAT*.

Fig.6 shows the *DPM-Distance-Penalty-SAT* procedure for solving general SAT instances. Since a large part of it is similar to that in Fig.4, we only explain the differences next.

Line 3 does not initialize temporary weights t_i as in *DPM-Trap-Avoidance-SAT* because we need not identify explicitly the locations of traps here. Such situations will be handled by $distance\text{-}penalty(x)$ automatically in the objective function.

Lines 7–8 maintain a queue on a fixed number of historical points. After a predefined number of flips, the algorithm inserts the current search point in the queue and deletes 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_s to be in the range [11, 19].

Lines 12–13 reduce α_i of all clauses by δ_d (= 1) when $\#_{Adjust}$ reaches threshold θ_2 (12 for *f*, 46 for *par*16, 7 for *g*, and 40 for *hanoi*4). These reductions are critical in our strategy because they help maintain the effect of $distance\_penalty(x)$ in L_d by keeping $\alpha^T h(x)$ in L_d in a suitable range, given that *distance_penalty* is in a fixed range that can be computed from w_s and θ_t. Without these reductions, *distance_penalty* will be relatively small when α becomes too large and has no significant effect in avoiding regions visited before.

As compared to *DPM-Trap-Avoidance-SAT* in Fig.4, the new approach is simpler and has less parameters to tune. We do not show frequency diagrams like those in Figs.2 and 5 because we do not identify traps explicitly, and the penalty function is different with the addition of *distance-penalty*. 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 *DPM-Trap-Avoidance-SAT*. However, the overall CPU time is actually much shorter for most benchmark problems tested because the new trap-avoidance strategy can avoid visiting the same regions more effectively.

We have applied *DPM-Distance-Penalty-SAT* to solve some hard, satisfiable SAT instances in the DIMACS archive. Table 3 shows that it can solve quickly *f*2000, *par*16-1-*c* to *par*16-5-*c*, *par*16-1 to *par*16-5, *hanoi*4 and *hanoi4-simple* with 100% success ratio. Comparisons to other algorithms are shows in Section 7.

7 Performance Comparisons

In this section, we compare our trap-avoidance strategies to some well-known global-search methods in solving

Table 3. Performance of *DPM-Distance-Penalty-SAT* in Solving Hard SAT Instances. Experiments were run on a 500MHz Pentinum-III computer with Solaris 7.

Problem ID	Succ. ratio	CPU (s)	# flips	Problem ID	Succ. ratio	CPU (s)	# flips
par16-1	10/10	101.7	1.3×10^7	par16-1-c	10/10	20.8	2,786,081
par16-2	10/10	154.0	2.1×10^7	par16-2-c	10/10	51.6	6,824,355
par16-3	10/10	76.3	9.8×10^6	par16-3-c	10/10	27.5	3,674,644
par16-4	10/10	83.7	1.1×10^7	par16-4-c	10/10	35.8	4,825,594
par16-5	10/10	121.9	1.5×10^7	par16-5-c	10/10	32.4	4,264,095
hanoi4	10/10	6,515	6.3×10^8	hanoi4-simple	10/10	9,040	1.1×10^9
g125-17	10/10	41.4	434,183	bw-large-a	10/10	0.10	6,176
g125-18	10/10	4.8	22,018	bw-large-b	10/10	1.55	67,946
g250-15	10/10	17.7	2,437	bw-large-c	10/10	72.36	1,375,437
g250-29	10/10	193.1	289,962	bw-large-d	10/10	146.28	1,112,332
f600	10/10	0.80	73,753	anomaly	10/10	0.00	259
f1000	10/10	3.21	285,024	medium	10/10	0.02	1,537
f2000	10/10	19.2	1,102,816	huge	10/10	0.19	10,320

Table 4. Performance Comparisons of *DPM-Trap-Avoidance-SAT*, *DPM-Distance-Penalty-SAT*, *WalkSAT/GSAT*, and *LSDL*[48] on Solving Some Hard SAT Instances. Our experiments were run on a 500-MHz Pentinum-III computer with Solaris 7. WalkSAT/GSAT was evaluated on an SGI Challenge with MPIS processor, model unknown. The timing results of *LSDL*, using two different strategies GENET and MAX, were collected on a SUN Sparc classic, model unknown. So far, *DPM-Distance-Penalty* has not found any solution to par32-?-c, as denoted by "–" in the table. ("NR" in the table stands for "not reported.")

Problem	*DPM-Distance Penalty*		*DPM-Trap Avoidance*		*WalkSAT/ GSAT*		*LSDL*	
							GENET	MAX
ID	Succ. ratio	Time (s)	Succ. ratio	Time (s)	Succ. ratio	Time (s)	Time (s)	Time (s)
par16-1	10/10	101.7	10/10	96.5	NR	NR	NR	NR
par16-2	10/10	154.0	10/10	95.7	NR	NR	NR	NR
par16-3	10/10	76.3	10/10	125.7	NR	NR	NR	NR
par16-4	10/10	83.7	10/10	54.5	NR NR	NR	NR	
par16-5	10/10	121.9	10/10	178.5	NR	NR	NR	NR
par16-1-c	10/10	20.8	10/10	28.8	NR	NR	NR	NR
par16-2-c	10/10	51.6	10/10	61.0	NR	NR	NR	NR
par16-3-c	10/10	27.5	10/10	35.3	NR	NR	NR	NR
par16-4-c	10/10	35.8	10/10	46.1	NR	NR	NR	NR
par16-5-c	10/10	32.4	10/10	44.6	NR	NR	NR	NR
par32-1-c	–	–	1/20	9,556	NR	NR	NR	NR
par32-2-c	–	–	1/20	113,714	NR	NR	NR	NR
par32-3-c	–	–	1/20	171,372	NR	NR	NR	NR
par32-4-c	–	–	1/20	128,537	NR	NR	NR	NR
f600	10/10	0.80	10/10	0.664	NR	35*	NR	NR
f1000	10/10	3.21	10/10	3.7	NR	1,095*	NR	NR
f2000	10/10	19.2	10/10	16.2	NR	3,255*	NR	NR
hanoi4	10/10	6,515	10/10	14,744	NR	NR	NR	NR
hanoi4-s	10/10	9,040	10/10	14,236	NR	NR	NR	NR
g125-17	10/10	41.4	10/10	144.8	7/10**	264**	282.0	192.0
g125-18	10/10	4.8	10/10	3.98	10/10**	1.9**	4.5	1.1
g250-15	10/10	17.7	10/10	12.9	10/10**	4.41**	0.418	0.328
g250-29	10/10	193.1	10/10	331.4	9/10**	1,219**	876.0	678.0
anomaly	10/10	0.00	10/10	0.01	NR	NR	NR	NR
medium	10/10	0.02	10/10	0.01	NR	NR	NR	NR
huge	10/10	0.19	10/10	0.18	NR	NR	NR	NR
bw-large-a	10/10	0.10	10/10	0.12	0.3***	NR	NR	NR
bw-large-b	10/10	1.55	10/10	2.43	22***	NR	NR	NR
bw-large-c	10/10	72.36	10/10	126.86	670***	NR	NR	NR
bw-large-d	10/10	146.28	10/10	254.80	937***	NR	NR	NR

*: Results from [15] for similar but not the same problems in the DIMACS archive
**: Results from [47]
***: Results from [48]

DIMACS/SATLIB benchmark instances. As most other methods do not report the number of flips, we omit this measure in our comparisons. The results demonstrate the robustness and effectiveness of DPM and trap avoidance in solving satisfiable SAT instances.

Table 4 compares our results to GSAT/WalkSAT and *LSDL*, two incomplete stochastic search methods in the area.

When compared to GSAT/WalkSAT, DPM with trap avoidance can solve many hard-to-satisfy problems, like *par*16-*, *par*32-1-*c* to *par*32-4-*c*, *hanoi*4, and *hanoi*4-*simple*, that cannot be solved by GSAT/WalkSAT. Our results on the *f*-class problems are around 100 times faster and have 100% success ratio from all ten randomly generated starting points on the *g*-class problems. In contrast, GSAT/WalkSAT were not 100% successful on *g*125-17 and *g*125-29. For problems in Blocksworld[49], DPM with trap avoidance also has significant improvements.

When compared to *LSDL*[50] on hard graph coloring problems, our algorithms outperform *LSDL* on *g*125-17 and *g*250-29, using 60% less CPU time, whereas *LSDL* is faster in solving the easier *g*125-18 and *g*250-15 problems. The latter happens because DPM incurs additional overhead in maintaining information used in trap avoidance. Hence, there is no apparent advantage of using trap avoidance for very simple problems. Moreover, *LSDL* uses a more efficient representation specific to graph coloring by representing a constraint violation as a link between two nodes.

Table 4 shows that *DPM-Trap-Avoidance-SAT* performs worse than *DPM-Distance-Penalty-SAT* on *par*16-, *hanoi*-, *g*-, and Blocksworld problems. *DPM-Trap-Avoidance-SAT* takes 50% more CPU times to solve *hanoi*4, *g*125-17, and *g*250-29. However, it performs better on the *f*-class problems that are randomly-generated 3-SAT problems. Further, it has found solutions to *par*32-1-*c* to *par*32-4-*c* that cannot be found by *DPM-Distance-Penalty-SAT*.

Finally, we compare our algorithms to Grasp[45], one of the best complete algorithms. Since Grasp performs the best on most DIMACS benchmarks[45] when compared to other complete methods, such as POSIT[51], CSAT[52], H2R[53] and DPL—a recent implementation of the Davis-Putnam procedure[2], we compare our results with respect to Grasp only.

Since Grasp is a complete method that can prove unsatisfiability, we expect it perform not as well on satisfiable problems. Table 5 shows that Grasp cannot solve problems in the *f*- and *g*- classes, and that DPM have consistently better performance on the problems tested.

8 Conclusions

We have demonstrated in this paper the theory of extended saddle points that provides a solid mathematical foundation for handling nonlinear discrete constraints and its application in solving large yet satisfiable SAT problems. Two important observations that can be made from our results are the significance of historical information found in a search and the way this information is represented and used.

Table 5. Performance Comparisons of *DPM-BASIC-SAT*, *DPM-Trap-Avoidance-SAT* and Grasp[45] on Some Typical DIMACS Benchmarks (The timing results of Grasp were collected on a SUN SPARC 5/85 computer. "–" stands for "not solved.")

Problem class	Success ratio	CPU (s)	GRASP	
			Succ. ratio	CPU (s)
aim50-	10/10	0.01**	10/10	0.4
aim100-	10/10	0.02**	10/10	1.8
aim200-	10/10	0.32**	10/10	10.8
ii8-	10/10	0.02**	10/10	23.4
ii16-	10/10	0.25**	10/10	10,311
ii32-	10/10	0.09**	10/10	7.0
ssa-	10/10	0.095**	10/10	6.5
f-	10/10	7.7*	–	–
g-	10/10	64.3*	–	–
par8-	10/10	0.25**	10/10	0.4
par16-	10/10	108*	10/10	9,844
hanoi-	10/10	7,778*	5/10	14,480

*: Average computed using values in Table 3
**: Average computed using values in Table 1

The first aspect in improving a search is to gather and maintain as much useful yet simple information as possible. In this paper, we collect information on traps in *DPM-Trap-Avoidance-SAT* and locations of points visited in the past in *DPM-Distance-Penalty-SAT*. Although our experimental results demonstrate that such information is useful, it is by no means complete. Other information that can be collected include the locations of basins, their sizes, profiles on the difficulty of satisfaction of each clause/constraint, the relationship of one clause to another in terms of constraint satisfaction, and the lengths of cycles that a trajectory has experienced.

The second aspect in improving a search is to represent efficiently the information collected in order for it to be recalled in the future. The information can be represented as a new objective, or new constraints, or both, in a penalty formulation. The secret, however, lies in the efficiency of implementation, since some of the unsolved yet feasible DIMACS benchmarks may require billions of flips, making it critical that each flip be very fast. Our experience in designing *DPM-Distance-Penalty-SAT* illustrates this point because early versions of our design were inefficient and performed poorly. Only careful re-implementations led to the current version that performs well.

In short, the keys to a successful SAT algorithm lie in the identification of useful historical information in a search and its efficient representation and recall.

References

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

[2] Barth P. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institute fur Informatik, 1995.

[3] Robinson J A. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 1965, pp.23–41.

[4] Genesereth M R, Nilsson N J. Logical Foundation of Artificial Intelligence. Morgan Kaufmann, 1987.

[5] Davis M, Putnam H. A computing procedure for quantification theory. *J. Assoc. Comput. Mach.*, 1960, 7: 201–215.

[6] Purdom P W. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 1983, 21: 117–133.

[7] Gu J, Wang W. A novel discrete relaxation architecture. *IEEE Trans. PAMI*, August 1992, 14(8): 857–865.

[8] Gu J. The UniSAT problem models (appendix). *IEEE Trans. PAMI*, Aug 1992, 14(8): 865.

[9] Gu J. Efficient local search for very large-scale satisfiability problems. *SIGART Bulletin*, January 1992, 3(1): 8–12.

[10] Gu J. Local search for satisfiability (SAT) problems. *IEEE Trans. Systems, Man, and Cybernetics*, 1993, 23(4): 1108–1129.

[11] Gu J. Global optimization for satisfiability (SAT) problems. *IEEE Trans. Knowledge and Data Engineering*, Jun 1994, 6(3): 361–381.

[12] Selman B *et al.* A new method for solving hard satisfiability problems. In *Proc. AAAI-92*, San Jose, 1992, pp.440–446.

[13] Selman B, Kautz H, Cohen B. Noise strategies for improving local search. In *Proc. 12th National Conf. Artificial Intelligence*, Seattle, WA, 1994, pp.337–343.

[14] Selman B, Kautz H. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proc. 13th Int. Joint Conf. Artificial Intelligence*, 1993, pp.290–295.

[15] Selman B *et al.* Local search strategies for satisfiability testing. In *Proc. 2nd DIMACS Challenge Workshop on Cliques, Coloring, and Satisfiability, Rutgers University*, 1993, pp.290–295.

[16] Hoos H H. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proc. Sixteenth National Conf. Artificial Intelligence*, 1999, pp.661–666.

[17] Gent I P, Walsh T. An empirical analysis of search in GSAT. *Journal of Artificial Intelligence Research*, 1993, 1: 25–37.

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

[19] Battiti R, Tecchiolli G. The reactive tabu search. *ORSA Journal on Computing*, 1994, 6(2): 126–140.

[20] Morris P. The breakout method for escaping from local minima. In *Proc. 11th National Conf. Artificial Intelligence*, Washington DC, 1993, pp.40–45.

[21] Frank J. Learning short-term weights for GSAT. In *Proc. 15th Int. Joint Conf. AI*, 1997, pp.384–391.

[22] Fukunaga A. Efficient implementations of SAT local search. In *Proc. 7th Int. Conf. Theory and Applications of Satisfiability Testing*, 2004.

[23] Van Hentenryck P, Michel L. Control abstractions for local search. In *Proc. the Ninth Int. Conf. Principles and Practice of Constraint Programming*, LNCS 2833, 2003, pp.65–80.

[24] Zhang L, Malik S. Towards symmetric treatment of conflicts and satisfaction in quantified Boolean satisfiability solver. In *Proc. 8th Int. Conf. Principles and Practice of Constraint Programming*. Springer Verlag, 2002.

[25] Moskewicz M *et al.* Chaff: Engineering an efficient SAT solver. In *Proc. 39th Design Automation Conf.*, 2001.

[26] Wah B W, Wang T. Simulated annealing with asymptotic convergence for nonlinear constrained global optimization. In *Proc. Principles and Practice of Constraint Programming*, Springer-Verlag, October 1999, pp.461–475.

[27] Folino G *et al.* Combining cellular genetic algorithms and local search for solving satisfiability problems. In *Proc. 10th IEEE Int. Conf. Tools with AI*, 1998, pp.192–198.

[28] Blair C E, Jeroslow R G, Lowe J K. Some results and experiments in programming techniques for propositional logic. *Computers and Operations Research*, 1986, 13(5): 633–645.

[29] Hooker J N. Resolution vs. cutting plane solution of inference problems: Some computational results. *Operations Research Letters*, 1988, 7: 1–7.

[30] Kamath A P, Karmarkar N K, Ramakrishnan K G, Resende M G C. Computational experience with an interior point algorithm on the satisfiability problem. *Annals of Operations Research*, 1990, 25: 43–58.

[31] Shi R C J, Vannelli A, Vlach J. An improvement on Karmarkar's algorithm for integer programming. *COAL Bulletin of the Mathematical Programming Society*, 1992, 21: 23–28.

[32] Wah B W, Shang Y. A discrete Lagrangian-based global-search method for solving satisfiability problems. In *Satisfiability Problem: Theory and Applications*, Ding-Zhu Du, Jun Gu, Panos Pardalos (eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1997, pp.365–392.

[33] Wah B W, Chang Y-J. Trace-based methods for solving nonlinear global optimization problems. *J. Global Optimization*, March 1997, 10(2): 107–141.

[34] Shang Y, Wah B W. Global optimization for neural network training. *IEEE Computer*, March 1996, 29(3): 45–54.

[35] Luenberger D G. Linear and Nonlinear Programming. Addison-Wesley, Reading, MA, 1984.

[36] Spellucci P. An SQP method for general nonlinear programs using only equality constrained subproblems. *Mathematical Programming*, 1998, 82: 413–448.

[37] Wah B W, Wang T. Constrained simulated annealing with applications in nonlinear constrained global optimization. In *Proc. Int. Conf. Tools with AI*, IEEE, Nov. 1999, pp.381–388.

[38] Chang Y J *et al.* Lagrangian techniques for solving a class of zero-one integer linear programs. In *Proc. Computer Software and Applications Conf.*, Dallas, Aug. 1995, IEEE, pp.156–161.

[39] Wah B, Chen Y X. Fast temporal planning using the theory of extended saddle points for mixed nonlinear optimization. *Artificial Intelligence*, (accepted for publication) 2005.

[40] Wu Z. Discrete Lagrangian methods for solving nonlinear discrete constrained optimization problems [Thesis]. Dept. Computer Science, Univ. Illinois, Urbana, IL, May 1998.

[41] Wah B W, Wu Z. The theory of discrete Lagrange multipliers for nonlinear discrete optimization. In *Proc. Principles and Practice of Constraint Programming*, Springer-Verlag, October 1999, pp.28–42.

[42] Aarts E, Korst J. Simulated Annealing and Boltzmann Machines. J. Wiley and Sons, 1989.

[43] Wu Z. The theory and applications of nonlinear constrained optimization using Lagrange multipliers [Dissertation]. Dept. Computer Science, Univ. Illinois, Urbana, IL, May 2001.

[44] Gent I P *et al.* Morphing: Combining structure and randomness. In *Proc. 16th National Conf. AI*, 1999, pp.654–660

[45] Marques-Silva J P, Sakalla K A. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, May 1999, 48(5): 506–521.

[46] Wu Z, Wah B W. Trap escaping strategies in discrete Lagrangian methods for solving hard satisfiability and maximum satisfiability problems. In *Proc. 1999 National Conf. Artificial Intelligence*, AAAI, July 1999, pp.673–678.

[47] Selman B. Private communcation, 1995.

[48] Kautz H, Selman B. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. the AAAI National Conf. AI*, 1996, pp.1194–1201.

[49] SATLIB suite. http://www.informatik.tu-darmstadt.de/AI/SATLIB.

[50] Choi K F M *et al.* A Lagrangian reconstruction of a class of local search methods. In *Proc. 10th Int. Conf. Artificial Intelligence Tools*, IEEE Computer Society, 1998.

[51] Freeman J W. Improvements to propositional satisfiability search algorithms [Dissertation]. Dept. Computer and Information Science, Univ. Pennsylvania, May 1995.

[52] Dubois O *et al.* SAT versus UNSAT. In *DIMACS Workshop on Satisfiability Testing*, New Brunswick, NJ, 1993.

[53] Pretolani D. Efficiency and stability of hypergraph SAT algorithms. In *Second DIMACS Implementation Challenge*, Johnson D S, Trick M A (eds.), Volume 26, American Mathematical Society, 1993, pp.479–498.

Benjamin W. Wah is currently the Franklin W. Woeltge Endowed Professor of Electrical and Computer Engineering and a professor of the Coordinated Science Lab. the Univ. Illinois at Urbana-Champaign, Urbana, IL. He received his Ph.D. degree in computer science from the Univ. California, Berkeley, CA, in 1979. Previously, he had served on the faculty of Purdue Univ. (1979–85), as a Program Director at the National Science Foundation (1988–89), as Fujitsu Visiting Chair Professor of Intelligence Engineering, Univ. Tokyo (1992), and McKay Visiting Professor of Electrical Engineering and Computer Science, Univ. California, Berkeley (1994). In 1989, he was awarded a University Scholar of the University of Illinois; in 1998, he received the IEEE Computer Society Technical Achievement Award; in 2000, the IEEE Millennium Medal; and in 2003, the Raymond T. Yeh Lifetime Achievement Award from the Society for Design and Process Science.

Wah's current research interests are in the areas of nonlinear search and optimization, multimedia signal processing, and computer networks.

Wah was the Editor-in-Chief of the *IEEE Trans. Knowledge and Data Engineering* between 1993 and 1996, and is the Honorary Editor-in-Chief of *Knowledge and Information Systems*. He currently serves on the editorial boards of *Information Sciences, Int. J. Artificial Intelligence Tools, J. VLSI Signal Processing, World Wide Web*, and *Neural Processing Letters*. He had chaired a number of international conferences and was the International Program Committee Chair of the IFIP World Congress in 2000. He has served the IEEE Computer Society in various capacities; among them include Vice President for Publications (1998 and 1999) and President (2001). He is a Fellow of the IEEE, AAAS, ACM, and the Society for Design and Process Science.

Zhe Wu was born on Jan. 4, 1974. He received his B.Sc. degree (June 1996) in the Dept. Special Class for Gifted Young from the Univ. Science and Technology of China and M.S. degree (May 1998) and Ph.D. degree (June 2001) from the Dept. Computer Science of the Univ. Illinois at Urbana-Champaign. He is currently working as a principal member of technical staff in the Oracle UDDI Server Team in Oracle Co. His main research interests are in optimization, signal processing, software engineering and computer networks.