The Tunneling Algorithm for Partial CSPs and Combinatorial Optimization Problems

Chris Voudouris VOUDCX@ESSEX.AC.UK
Edward Tsang EDWARD@ESSEX.AC.UK
Department of Computer Science,
University of Essex,
Colchester, C04 3SQ, UK

Abstract

Constraint satisfaction is the core of a large number of problems, notably scheduling. Because of their potential for containing the combinatorial explosion problem in constraint satisfaction, local search methods have received a lot of attention in the last few years. The problem with these methods is that they can be trapped in local minima. GENET is a connectionist approach to constraint satisfaction. It escapes local minima by means of a weight adjustment scheme, which has been demonstrated to be highly effective. The tunneling algorithm described in this paper is an extension of GENET for optimization. The main idea is to introduce modifications to the function which is to be optimized by the network (this function mirrors the objective function which is specified in the problem). We demonstrate the outstanding performance of this algorithm on constraint satisfaction problems, constraint satisfaction optimization problems, partial constraint satisfaction problems, radio frequency allocation problems and traveling salesman problems.

1. Introduction

Local search methods being general approximation algorithms have been used widely to solve optimization problems. A recent application domain of local search is the *Constraint Satisfaction Problem* (CSP) (Tsang, 1993). A CSP involves assigning values to a set of variables satisfying a set of constraints. One of the best known work in applying local search to constraint satisfaction is the *heuristic repair method* which uses the *min-conflicts heuristic* (Minton et. al. 1992). A min-conflict based move (also mentioned as repair or "flip") is realized by re-assigning to a variable (which in current assignment violates some constraints) the value which violates the least number of constraints.

A local search approach to constraint satisfaction treats a CSP as an optimization problem. The objective function, which is to be minimized, is the number of constraints being violated. A typical local search method assigns an arbitrary value to each variable in the CSP. Then it proceeds iteratively to reduce the number of constraint violations by re-assigning values to variables, using heuristics such as min-conflict. This iterative improvement of the number of unsatisfied constraints leads either to a solution to the CSP or to a local minimum where some constraints are still being violated but no further improvement is possible by changing the value of any of the variables. Various algorithms based on local search incorporate schemes that enhance local search to escape local minima or avoid them (Wang & Tsang, 1991; Selman & Kautz, 1993; Morris, 1993; Selman et al. 1994).

GENET (Wang & Tsang, 1991; Davenport et al. 1994) is a connectionist approach to constraint satisfaction with a basic operation that resembles the min-conflicts heuristic. Basically a CSP is represented by a network in which the nodes represent possible assignments

to the variables and the edges represent constraints. One of the innovations in GENET is the use of and manipulation of weights assigned to the edges (constraints). All edges are inhibitory connections which have weights initialized to -1. GENET will continuously select assignments which receive the least inhibitory input (which roughly means violating the least number of constraints). The operation of the network is designed in such a way that will ensure its convergence to some states[1] , which could be solutions or local minimum (in terms of number of constraints violated). Each time the network converges to a local minimum, the weights associated to the violated constraints are decreased, and the network is then allowed to converge again. Such convergence-learning cycles continue until a solution is found or a stopping condition is satisfied.

Some GENET models perform remarkably well on CSPs in which the aim is to find any solution which satisfies all the constraints (Davenport et al. 1994). However, in some applications, finding just any solution is not good enough: one is expected to find optimal or near-optimal solutions subject to one or more optimization criteria. Besides, when a CSP is insoluble, partial solutions that minimize costs associated to unsatisfied constraints are required. This paper reports a GENET model which tackles such problems and the *tunneling algorithm* this model is based upon. It is probably worth mentioning that backtracking constraint satisfaction algorithms developed over the years rarely address these issues and when they do, the combinatorial explosion problem[2] prevents them from solving most real life problems which usually involve large numbers of variables.

Local search methods, including the *tunneling algorithm* which we are going to describe in this paper, cast CSPs as optimization problems where the optimization criterion is the number of constraint violations. Partial CSPs and Constraint Satisfaction Optimization Problems (CSOPs), which will be further examined later in this paper, fit well under this paradigm. We shall further show that the tunneling algorithm is in fact applicable to other traditional combinatorial optimization problems such as the Traveling Salesman Problem.

2. Basic Idea Of The Tunneling Algorithm And Related Methods

GENET's mechanism for escaping from local minima resembles reinforcement learning (Barto et al. 1983). Basically, patterns in a local minimum are stored in the constraint weights and are discouraged to appear thereafter. For this reason, the mechanism was named "learning". GENET's learning scheme can be viewed as a method to transform the objective function (i.e. the number of constraint violations) so that a local minimum gains an artificially higher value. Consequently, local search will be able to leave the local minimum state and search other parts of the space.

In fact, the idea of modifying the function to be optimized is not unique to GENET. Algorithms in optimization have emerged during the past decade based on exactly the same principle. These algorithms are known as *tunneling algorithms* (Zhigljavsky, 1991; Levy & Montalvo, 1985). The modified objective function is called the *tunneling function*. This function allows local search to explore states which have higher costs around or further away from the local minimum. The direction of exploration depends on the way modifications are introduced in the tunneling function. In tunneling, it is not only important to escape from the local minimum (some other techniques achieve the same effect) but also to select a tunneling direction (see Section 9.1) which has a good chance of arriving at states with lower costs

[1] To be precise, a number of models have been developed in GENET, some of which guarantee convergence. In models such as the Stable1-Sideway Model which will be mentioned later in this paper, convergence is defined as the network staying in the same state in two consecutive iterations.

[2] Accepting possible constraint violations vastly increases the search space that must be considered by a complete search method.

according to the objective function. Furthermore, the strength of tunneling to overcome hills on the way is to be determined (see Section 9.2).

The tunneling algorithm perceived in the way described above, provides an alternative to two other popular techniques which also allow uphill moves, namely

- Simulated Annealing (Kirkpatrick et al. 1983; Aarts & Korst, 1989) and
- Tabu Search (Glover, 1989; Glover, 1990).

Simulated annealing accepts, in a limited way, transitions which lead to an increase of values according to the objective function. This is true for tunneling too. The difference is that simulated annealing never modifies the function to be optimized, and therefore a local minimum will always remain as a local minimum.

In tabu search, the escape of local minima is accomplished by continuously maintaining a list of tabu states or actions: this prevents the algorithm from staying in and revisiting local minima. Since there is no limit on what form the tabu list takes and how it is updated, one can see tunneling algorithms (as well as a large number of other algorithms) as a class of tabu search.

3. Partial Constraint Satisfaction Problem

For convenience, we shall first define some terminology. The assignment of a value to a variable is called a *label*. The label which involves the assignment of a value v to the variable x (where v is in the domain of x) is denoted by the pair $<x,v>$. A simultaneous assignment of values to a set of variables is called a *compound label* and is represented as a set of labels, denoted by $(<x_1,v_1>,<x_2,v_2>,...,<x_k,v_k>)$. A *complete compound label* is a compound label which assigns a value to every variable in the CSP. A network *state* in GENET represents a complete compound label. Naturally, every state in the network represents a candidate solution for a CSP.

A *Partial Constraint Satisfaction Problem* (PCSP) is a CSP in which one is prepared to settle for partial solutions — solutions which may violate some constraints or assignments of values to some, but not all variables — when solutions do not exist (or, in some cases, cannot be found) (Freuder & Wallace, 1992; Tsang, 1993). This kind of situation often occurs in applications like industrial scheduling where the available resources are not enough to cover the requirements. Under these circumstances, partial solutions are acceptable and a problem solver has to find the one that minimizes an objective function.

The objective function is domain-dependent and may take various forms. In one of its simplest forms, the optimization criterion may be the number of the constraint violations. For more realistic settings, some constraints may be characterized as "hard constraints" and they must be satisfied whilst others, which are referred to as "soft constraints", may be violated if necessary. Moreover, constraints may be assigned violation costs which reflect their relative importance. Following (Tsang, 1993), we define the Partial Constraint Satisfaction Problem formally as follows:

Definition 1-1:

A *partial constraint satisfaction problem* (PCSP) is a quadruple:

$$(Z, D, C, g)$$

where

- $Z = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables,
- $D = \{D_{x_1}, D_{x_2}, \ldots, D_{x_n}\}$ is a set of finite domains for the variables in Z,
- $C = \{c_1, c_2, \ldots, c_m\}$ is a finite set of constraints on an arbitrary subset of variables in Z,
- g is the objective function which maps every compound label to a numerical value.

The goal in a PCSP is to find a compound label (partial or complete) which optimizes (minimizes or maximizes) the objective function g. Given the above definition, standard CSPs and *Constraint Satisfaction Optimization Problems* (CSOPs) (where optimal solutions are required in CSPs, see Tsang, 1993) can both be cast as PCSPs.

Versions of branch and bound and other complete methods have been suggested for tackling PCSPs (Freuder & Wallace, 1992; Wallace & Freuder, 1993). But complete algorithms are inevitably limited by the combinatorial explosion problem. This motivates our development of GENET models for tackling PCSPs. In the following sections, we shall introduce a general form of an optimization function (g) which allows us to describe CSPs, CSOPs as well as PCSPs.

4. Tunneling In The Basic GENET Models

As mentioned before, the learning mechanism used in GENET can be seen as a form of tunneling. In GENET, the cost function is the number of constraint violations. A solution of a CSP is a network state in which no constraint is violated. To modify this function during the search, weights are defined for each of the constraints. These weights are initialized to -1. The cost associated to a constraint c_k at a network state S is defined as follows:

$$c_k(S) = \begin{cases} -w_k & ,\mathrm{c_k \text{ is violated in S}} \\ 0 & ,\mathrm{c_k \text{ is not violated in S}} \end{cases} \quad \text{(Eq. 1)}$$

where w_k is the weight associated to the constraint c_k. In other words, if the constraint c_k is violated, then a cost of $-w_k$ is incurred; otherwise no cost is incurred. The cost function that GENET minimizes can be summarized as follows:

$$g(S) = \sum_{c_k \in C} c_k(S) \quad \text{(Eq. 2)}$$

where C is the set of constraints in the problem. If all the weights are equal to -1, as would be the case when the search starts, then (Eq. 2) gives the number of unsatisfied constraints.

If the network converges to a local minimum S_*, then the weights are decreased. Since GENET always makes moves which improve g, decreasing the weights allows it to escape from S_* to states which have lower cost. We call (Eq. 2) the *tunneling function*. The tunneling algorithm controls the tunneling function by modifying appropriately its terms using weights or penalties which will be described later in the paper. The success of a tunneling algorithm in an application is to a large extent determined by the way the tunneling function is defined and the mechanism for altering this function at local minima states. GENET's learning mechanism has been demonstrated to be effective in guiding the search towards solutions in satisfiability problems. In the following section we shall define a general cost function for PCSPs and present a mechanism (which generalizes from GENET's learning mechanism) for manipulating the tunneling function.

5. Objective and Tunneling Function for PCSPs

The tunneling function must reflect the characteristics of the objective function of the problem (Zhigljavsky, 1991). For CSPs, the objective function is simply the number of constraint violations. Above (Eq. 2) we have defined the tunneling function of a network state in CSPs, by weighting constraint violations. This scheme carries a limitation. It assumes that constraints are of equal importance and therefore all the initial weights are set to -1. To refine and extend this scheme to PCSPs, we define a **positive** cost for each constraint and use penalties to achieve the modifications formerly accomplished by the weighting scheme. Moreover, we define a generic cost function which incorporates *constraint costs* as well as *assignment costs*. In the following sections, we shall define these costs in detail and explain how the added penalties transform the objective function to the tunneling one.

5.1 Constraint Costs

In (Eq. 1), we made the implicit assumption that the cost of violating a constraint reflects the weight associated to it in the network only. This is not always true. In an application, a constraint violation may reflect the importance of that constraint in the problem specification. For example, violating a hard constraint may incur an unacceptably high cost and violating a soft constraint may incur a cost which reflects the importance of that constraint. If r_k is the cost of violating the constraint c_k defined in the problem specification (it is also referred to as relaxation cost) then the cost associated to c_k in the state S is defined below:

$$c_k(S) = \begin{cases} r_k & ,\mathrm{c_k} \text{ is violated in S} \\ 0 & ,\mathrm{c_k} \text{ is not violated in S} \end{cases} \qquad \text{(Eq. 3)}$$

We will refer to this cost as the *primary constraint cost*. The indicator type function of the constraint cost is modified to give its tunneling version by adding a penalty term p_k to the primary cost. The *tunneling constraint cost* corresponding to (Eq. 3) is defined as follows:

$$c_k^T(S) = \begin{cases} r_k + p_k & ,\mathrm{c_k} \text{ is violated in S} \\ 0 & ,\mathrm{c_k} \text{ is not violated in S} \end{cases} \qquad \text{(Eq. 4)}$$

The penalty p_k is set to 0 at the beginning of the search and its value is increased by the tunneling algorithm when the network reaches local minimum situations[3]. The amount by which the penalty is to be increased in p_k is vital to the functioning of the algorithm, and will be discussed later in this paper. When p_k is increased, the cost level for the set of all possible states in which the constraint is violated rises with respect to other states. This will help the network to escape from local minimum situations.

5.2 Label Costs

In some applications, the assignment of different values to a variable incurs different costs. For example, different machines may have different costs to run; when modeled properly, labels which represent the use of precious resources should have higher costs. Utilities may

[3] In all our work so far, the penalty is only increased by the tunneling algorithm, but there is nothing to stop it being decreased in appropriate situations.

also be modeled as negative costs; for example, assigning different staff to a job may generate different amount of income.

When the above costs (or negative utilities) are linear, they are modeled by the *label costs* in our generic cost function. Label costs are to be minimized along with the constraint costs. Let x_i be a variable and its domain $D_{xi}= \{v_{i1},v_{i2},...,v_{im}\}$. As mentioned above, a label is a variable - value pair $<x_i,v_{ij}>$. Each label $<x_i,v_{ij}>$ is assigned a label cost a_{ij} in N. This label cost expresses the cost of assigning the value v_{ij} to the variable x_i. Label costs can model any separable objective function of the form:

$$f = \sum f_i(x_i) \quad \text{(Eq. 5)}$$

The *primary label cost* of the label $l_{ij}\equiv<x_i,v_{ij}>$ reflects the cost of assigning v_{ij} to x_i specified in the problem. The primary label cost of l_{ij} in the state S is given again by an indicator function:

$$l_{ij}(S) = \begin{cases} a_{ij} & ,\langle x_i, v_{ij} \rangle \in S \\ 0 & ,\langle x_i, v_{ij} \rangle \notin S \end{cases} \quad \text{(Eq. 6)}$$

The *tunneling label cost* corresponding to (Eq. 6) includes the penalty term p_{ij} added to the primary label cost a_{ij}:

$$l_{ij}^T(S) = \begin{cases} a_{ij} + p_{ij} & ,\langle x_i, v_{ij} \rangle \in S \\ 0 & ,\langle x_i, v_{ij} \rangle \notin S \end{cases} \quad \text{(Eq. 7)}$$

Note that this model can be extended to allow label costs to be associated to compound labels rather than labels, in which case (Eq. 6) and (Eq. 7) only need to be changed slightly.

5.3 A Generic Objective Function

The objective function of PCSPs can be defined as the sum of constraint and label costs, as mentioned in Sections 5.1 and 5.2:

$$g(S) = \sum_{i=1}^{|Z|} \sum_{j=1}^{|D_{x_i}|} l_{ij}(S) + \sum_{k=1}^{|C|} c_k(S) \quad \text{(Eq. 9)}$$

The tunneling function is basically (Eq. 9) with the terms being replaced by the corresponding tunneling costs, as shown below:

$$g^T(S) = \sum_{i=1}^{|Z|} \sum_{j=1}^{|D_{x_i}|} l_{ij}^T(S) + \sum_{k=1}^{|C|} c_{ij}^T(S) \quad \text{(Eq. 10)}$$

Since CSPs and CSOPs can be seen as PCSPs (CSPs are treated as optimization problems in GENET), the above objective and tunneling functions can be used in these problems.

As mentioned above, the tunneling function may be modified during search by the tunneling algorithm in order to force local search to escape from the local minimum. This is accomplished by increasing the penalties (to be explained in Section 9 later) of the terms included in (Eq. 10) that contribute to the objective function (i.e. selected labels, unsatisfied

constraints) in the local minimum. Some readers may have noticed that the tunneling and the objective function have a form similar to the objective function of the *Assignment Problem* (Papadimitriou & Steiglitz, 1982). In addition, the incorporation of constraints in the objective function is analogous to *penalty functions* and *lagrange multipliers* (Whittle, 1971) used in optimization under constraints.

6. Local Search In PCSPs

In the preceding section, the objective and tunneling functions were defined for PCSPs. It is the responsibility of the local search procedure to minimize these functions. PCSPs can be seen as multi-dimensional problems where each variable defines a dimension. One way to perform local search is to consider one dimension at a time. The neighborhood structure is formed by the values of the variable. Although parallelism is possible (Wang & Tsang, 1992), one-dimensional minimization must be conducted at a fine grain level. One important issue is to determine the scheme to be used for choosing the dimension (i.e. variable) to minimize in each step. Min-Conflicts Hill Climbing (MCHC) (Minton et al. 1992) chooses one variable at each iteration while GENET allows all the variables to update their values in each cycle; in simulation, every variable is examined once in each cycle.

In the algorithms presented in this paper, the variables are examined in a round robin fashion. In other words, all the variables are potentially updated in each iteration. The ordering of the variables in each iteration can be either static or randomized. In our tests, static orderings tend to give better results[4] than random orderings, though this is not always the case. Another important parameter in our algorithms is the incorporation of "sideways" moves, i.e. allowing a change of value for a variable even when doing so does not reduce the cost (a move is never made if it increases the cost). Our tests show that sideways moves may improve the performance of local search significantly in certain problems (this agrees with the results in Selman et al. 1992) but allowing sideways moves may result in the network not converging. Therefore, we incorporate a limited sideways scheme where sideways moves are allowed locally as far as the overall objective function changes value. The network is considered to have reached a local minimum when the value of the objective function stays the same for two consecutive iterations of the algorithm.

```
PROCEDURE GENERATE(Z, D, g, State_i, State_i+1)
BEGIN
        State ← State_i;
        FOR each variable x in Z DO  // local search
        BEGIN
                State ← State - {<x,v_i>};
                FOR each value v in Dx DO
                BEGIN
                        g_v ← g(State + {<x,v>});
                END
                BestSet ← set of values with minimum g_v;
                v_i+1 ← random value in BestSet; // i.e. sideways moves are allowed
                State ← State + {<x,v_i+1>};
        END
        State_i+1 ← State;
END
```

Figure 1. The GENERATE procedure in pseudocode.

[4] Under static orderings local search descends faster to a local minimum, something which is desirable in tunneling.

The pseudocode in Figure 1 presents the GENERATE procedure which from the current state S_i generates the next state S_{i+1} in the tunneling algorithm. A state is defined by the set of selected labels for the variables (as described in Section 3). Note that this procedure is replaced by other local search procedures when the tunneling algorithm is applied to other combinatorial optimization problems (such as Traveling Salesman Problem).

The complexity of GENERATE is $O(n\alpha)$ where n is the number of variables and α the maximum domain size. The efficiency of GENERATE can be improved by various schemes. For example, instead of evaluating the overall objective function (or tunneling function, depending on what we minimize), we may have local evaluations that refer only to those terms of g that are affected by the variable. The tunneling algorithm calls GENERATE iteratively. Two variations of the algorithm, namely *One-Stage Tunneling* (1ST) and *Two-Stage Tunneling* (2ST) have been developed, which will be described below. GENET's learning scheme is similar to the mechanism for modifying the tunneling function in the 1ST algorithm.

7. One-Stage Tunneling (1ST)

One-stage tunneling uses nothing but the tunneling function (Eq. 10) presented in Section 5.3. Search starts from an arbitrary assignment of values to the variables and all the penalties are set to 0. The GENERATE procedure is invoked repeatedly until the cost of the objective function (not that of tunneling function) remains the same for two consecutive iterations. A local minimum is then deduced[5] and the penalties increased appropriately (this will be explained later). Then GENERATE procedure is invoked repeatedly again until the next local minimum is reached. The outer loop repeats until a termination criterion is satisfied. In the general case, the criterion is either a maximum number of iterations or a time budget. In CSPs, the algorithm will also be stopped when a solution is found. The 1ST algorithm in pseudocode is shown in Figure 2.

```
PROCEDURE 1ST(Z, D, g, g^T, S*)
BEGIN
    k ← 0; S_k ← arbitrary assignment of values to variables;
    best_cost_so_far ← g(S_k);
    best_state_so_far ← S_k;
    REPEAT
        REPEAT
            Local_Minimum ← False;
            GENERATE(Z, D, g^T, S_k, S_k+1);
            IF (best_cost_so_far > g(S_k+1)) THEN
                BEGIN
                    best_cost_so_far ← g(S_k+1);
                    best_state_so_far ← S_k+1;
                END
            IF (g(S_k)=g(S_k+1)) THEN Local_Minimum ← True;
            k ← k+1;
        UNTIL (Local_Minimum)
        INCREASE_PENALTIES(S_k)^6 ;
    UNTIL (Stopping_Criterion);
    S* ← best_state_so_far;
END
```

Figure 2. The pseudocode for 1ST.

[5] Other schemes for detecting local minima are also possible. The scheme based on the value of the objective function, which is presented here, makes use of sideways moves avoiding cycling at the same time.
[6] The INCREASE_PENALTIES procedure is given in Section 9.2.

Readers may notice that in the above algorithm, input to the GENERATE procedure is the tunneling function g^T. The tunneling function may have a different cost surface from the objective function due to the penalties incorporated to constraints and labels. After the first local minimum is encountered, local search applies to the altered cost surface rather than the original one. This may cause problems since as the algorithm proceeds, the tunneling function may become very different from the objective function, and therefore the search may be misguided.

In particular, we faced problems with 1ST when applied to CSPs with a specific type of constraint. The penalties introduced were over-distorting the cost surface and 1ST was unable to find the solution for some over-constrained problems. Recently, the same problem was observed in two specially structured TSPs where 1ST was not able to improve significantly from the first local minimum and was driving off good configurations.

The answer to these problems was given by the Two-Stage tunneling algorithm (2ST). The key to 2ST is to periodically replace the tunneling function with the primary objective function. Such an action grounds the tunneling algorithm and keeps it concentrated to the promising states. This second algorithm is inspired by traditional optimization research, e.g. see (Zhigljavsky, 1991).

8. Two-Stage Tunneling (2ST)

The 2ST algorithm has two stages, namely *minimization* and *tunneling*. During the minimization stage, local search minimizes the objective function; during the tunneling stage, the tunneling function is minimized. The algorithm starts with the minimization stage. In a local minimum, the algorithm increases the penalties and changes its stage.

```
PROCEDURE 2ST(Z, D, g, g^T ,S^*)
BEGIN
        k ← 0;  Stage ← Minimization;
        S_k ← arbitrary assignment of values to variables;
        best_cost_so_far ← g(S_k);
        best_state_so_far ← S_k;
        REPEAT
                REPEAT
                        Local_Minimum ← False;
                        IF (Stage is Minimization) THEN GENERATE(Z, D, g, S_k, S_k+1);
                        ELSE  IF  (Stage is Tunneling)  THEN GENERATE(Z, D, g^T, S_k, S_k+1);
                        IF (best_cost_so_far > g(S_k+1) THEN
                                BEGIN   // change to minimization to reach the base of local minimum
                                        Stage ← Minimization;
                                        best_cost_so_far ← g(S_k+1);
                                        best_state_so_far ← S_k+1;
                                END
                        IF (g(S_k)=g(S_k+1)) THEN Local_Minimum ← True;
                        k ← k+1;
                UNTIL (Local_Minimum)
                INCREASE_PENALTIES(S_k);
                IF (Stage is Minimization) THEN Stage ← Tunneling;
                ELSE IF (Stage is Tunneling) THEN Stage ← Minimization;
        UNTIL (Stopping_Criterion);
        S^* ← best_state_so_far;
END
```

Figure 3. The pseudocode for 2ST.

The aim of the tunneling stage is to help local search to escape local minima. However, since it is the objective function that matters, the algorithm uses the objective function once it has escaped from a local minimum. Figure 3 gives the pseudocode for two-stage tunneling.

As we shall explain in the next section, care has been taken to ensure that the penalties increased by the procedure INCREASE_PENALTIES are high enough to allow the algorithm to escape from the local minimum that the algorithm is currently in. If the amount of penalty is too small, the algorithm will have to change between the minimization and tunneling stages a number of times before escaping the local minimum.

9. The Mechanism of Penalties

Both the algorithms described rely on the INCREASE_PENALTIES procedure to modify the tunneling function in such a way that would enable them to escape from the local minimum state. The modification increases penalties for some of the terms included in the tunneling function which are present in the local minimum. In this section, we shall describe this mechanism in detail. As it has been mentioned before, these terms are either label costs or constraint costs.

To recapitulate, each term that contributes to the tunneling function has a penalty assigned to it. At the beginning of the search all the penalties are equal to 0 and the tunneling function is equivalent to the objective one. Each time the local search algorithm descends to a local minimum, the procedure INCREASE_PENALTIES is called to increase certain penalties. The main objective of doing so is not just to raise the cost surface in the local minimum but also to shape it in such a way that the algorithm will follow a trajectory towards states with lower costs. Therefore, it is preferable to penalize constraints and labels with higher costs first[7] . The penalty-updating mechanism must decide on the following two issues:

- The terms to be penalized;
- The amount of penalty to be increased.

To make these decisions, the tunneling algorithm uses a simple heuristic based on frequency counts and cost information. This heuristic, which will be described in the following sections, has been proved to be effective in problems studied so far.

9.1 Selecting the Terms to be Penalized (FCR Heuristic)

The selection of terms to be penalized is based on two criteria. The first is the contribution (i.e. cost) of a term (representing a constraint or label cost) to the objective function. The second is the number of times (absolute frequency) that this term has been penalized in the past. For each term (constraint or label), the *Frequency to Cost Ratio* (FCR) is computed by the number of times a term has been penalized in the past divided by the cost of the term:

$$FCR = \frac{Frequency}{Cost} \qquad \text{(Eq. 11)}$$

[7] A plausible assumption is that states in which high cost terms do not contribute to the objective function are likely to be better. By penalizing the high cost terms, we enable local search to explore these more promising states.

A set *MinFCR* is constructed in the local minimum state S_*. This set includes the terms that

- contribute positively to the objective function in S_* (i.e. the terms which represent the costs of those violated constraints and selected labels); and
- have the minimum FCR value.

The *PenalizeSet* is a subset of *MinFCR* which contains the elements of *MinFCR* with the highest primary cost. Thus, the overall selection procedure employs a min-max like operation where we penalize the elements with the maximum cost amongst those with the minimum FCR value.

Initially, the FCR values are all equal to 0. So the selection is made purely on the primary costs. After the tunneling function has been modified, frequency counts are increased for the penalized terms. Since frequency counts are divided by the cost, FCR values increase faster for low cost elements than for high cost elements and therefore high cost elements get more chances to be penalized than the low cost elements. Nevertheless, low cost elements will still be penalized if the high cost elements have been penalized enough number of times. In CSPs, where all the constraints have the same violation cost and considered hard, we can simply penalize all the constraints violated in the local minimum without considering FCR values, which is what the basic GENET models do. In CSOPs, this mechanism allows us to select alternative labels when no constraints are violated — which means after a solution has been reached, the algorithm is given a chance to find better solutions.

Ultimately, each term considered in the selection process corresponds to a tunneling direction. Penalizing the term will propel local search towards that direction. Essentially, the tunneling direction points to states where the corresponding term has no contribution to the cost function. To diversify the search, it is better to follow tunneling directions not followed in past local minima. This results in a sequence of 'conjugate' tunneling directions. Finding the global optimum may require the tunneling directions to be followed several times over several local minima. FCR aims to do that by providing a cost-regulated cycling of tunneling directions starting with the most promising ones and devoting effort proportionally to the cost of terms. Finally, penalizing all the terms in the *PenalizeSet* substantially speeds up the algorithm since propulsion is applied to more than one direction.

9.2 Amount of Penalty

When a term has been selected for penalizing, the amount of penalty to be added is determined on the basis of information gathered by the GENERATE procedure presented in Section 5. The gathering of information is not included in the pseudocode in order to keep it simple. In reality, for each variable x_i, the algorithm records the difference (Δg_v) between the minimum value (g_v) and second minimum value returned by the objective function (or tunneling function, depending on the input of GENERATE) for all possible labels for x_i. This difference has to be overcome by modifying the tunneling function if any change in the assignment of x_i is to occur at all. The main aim of keeping these differences is that they provide a rough estimation of the relative height of the shortest hills that surround the local minimum.

In general, PCSPs are problems where arbitrarily small or large discrepancies in cost may occur between the local minimum and its surrounding states. For example, consider one extreme where a solution is found which only incurs a small assignment cost and all the surrounding states violate hard constraints of the problem. On the other hand, situations are likely where the cost of soft constraints violated in the local minimum is just slightly less than the total cost of violations incurring in the neighboring states. Therefore, the differences gathered provide invaluable information to determine the scale of penalty amounts to be considered.

Another consideration comes from the nature of tunneling algorithm. In particular, the penalty amount to be added to a constraint should be proportional to the primary cost of a constraint such that the properties of the primary cost surface are carried along to the distorted surface. Similarly, if the term which has been selected to be penalized represents a label, the amount of penalty to be added should reflect the primary cost of that label according to the objective function.

The tunneling algorithm is built upon the binary GENET models, which have been demonstrated to be successful for binary CSPs. In the binary models, all constraints have primary costs of 1. When a constraint is being violated in a local minimum, the cost of the violated constraint is increased by 1. The landscapes occurring in binary GENET are relatively smooth since the surrounding hills have heights that are a few times multiple of the primary constraint cost. The penalty amount in that case can be equal to the primary cost as it is in binary GENET. In the first extreme case mentioned above, it is obvious that such a scheme is to fail in PCSPs since the surrounding hills may have cost hundreds or thousands of times the cost of the local minimum. If the penalty is only to increase by an amount equal to the primary cost, a large number of iterations could be needed before a change happens in the GENET network. Thus to determine the penalty amount, we have to take into account both the cost of the term and the differences Δg_v gathered during the GENERATE procedure.

One potentially useful measure is the maximum difference Δg_v. This measure gives the maximum discrepancy between the local minimum and the closest neighboring states in all local search directions. An alternative measure is the minimum difference Δg_v. This measure can be term dependent since each term is associated to a subset of the variables (e.g. constraint) and therefore the minimum is taken only amongst these variables. These two measures combined with the cost of the term give the following two candidate formulas for the penalty amount.

$$PenaltyAmount = \max\left\{\text{cost}, \max\left\{\Delta g_v\right\}\right\} \quad \text{(Eq. 12)}$$

and

$$PenaltyAmount = \max\left\{\text{cost}, \min\left\{\Delta g_v\right\}\right\} \quad \text{(Eq. 13)}$$

The first formula (Eq. 12) guarantees that a change in the assignment of at least one variable will occur, even if the term is the only that is penalized. The second formula (Eq. 13) guarantees the same but it does not assure that the term will stop contributing to the cost function since that may be subject to changing the value of more than one variable. So which is the best formula? A first choice could be (Eq. 12) which needs less computation since $\max\left\{\Delta g_v\right\}$ has to be computed only once for all terms. However, the amount of additional penalty defined in (Eq. 12) is more than enough to make the network change state and that penalty excess may distort the landscape of the search space unnecessarily, and consequently mislead the search. In the experiments, we used both formulas. It appears to be the case that (Eq. 12) performs better in landscapes with very steep hills while (Eq. 13) is more effective where smoother landscapes are expected.

The pseudocode in Figure 4 summarizes the procedure for increasing the penalties where $\texttt{Penalty}_\texttt{m}$ and $\texttt{Frequency}_\texttt{m}$ are set to 0 for all the terms in the beginning of 1ST or 2ST and the `PenaltyAmount` is given either by (Eq. 12) or (Eq. 13).

```
PROCEDURE INCREASE_PENALTIES(State)
BEGIN
        T_labels ← set of labels in State with cost > 0;
        T_constraints ← set of constraints that are violated;
        T ← T_labels + T_constraints;
        MinFCR ← set of elements in T with minimum FCR value;
        PenalizeSet ← elements in MinFCR with highest cost;
        FOR each element m in PenalizeSet DO
                BEGIN
                        Penalty_m ← Penalty_m + PenaltyAmount;
                         // PenaltyAmount is defined in (Eq. 12 or 13)
                        Frequency_m ← Frequency_m +1;
                END
END
```

Figure 4. The INCREASE_PENALTIES procedure in pseudocode.

10. Experimentation and Results

Like most other stochastic search methods, the tunneling algorithm cannot guarantee to find the optimal solution. The main target of the experimentation presented here was to provide practical evidence that the algorithm is capable of finding the optimal or near optimal solutions reliably, subject to limited resources and stopping criterion. The objectives of the experiments were to determine whether:

- tunneling advances over GENET's performance in CSPs;
- tunneling is applicable to randomly generated PCSPs;
- tunneling can face the challenge of real world CSOPs and PCSPs; and
- tunneling can be used for tackling other combinatorial optimization problems.

To accomplish the objectives listed above, we conducted experiments on a diversified set of problems, which includes problems from the following categories:

- randomly generated general constraint satisfaction problems
- hard graph coloring problems in the literature
- randomly generated binary partial CSPs
- real life CSOPs and PCSPs
- traveling salesman problems

For CSPs, comparisons were carried out with original GENET, Min-Conflicts Hill Climbing (MCHC) (Minton et al. 1992) and GSAT (Selman & Kautz, 1993). Optimization problems were taken from the literature and results were compared with the best solutions known so far whenever they are available. Whenever the optimal solutions are known, the average solution errors of the tunneling algorithm are given.

10.1 Random General CSPs

Davenport *et al*. (1994) reported results for GENET on random general CSPs involving *atmost* constraints. The *atmost* constraint type is extended from CHIP (Dincbas et al. 1988). It is useful for modeling certain constraints imposed on resources in scheduling problems. Given a set of variables *Var* and a set of values *Val*, the *atmost(N, Var, Val)* constraint specifies that no

more than N variables from Var may take values from Val. The set of problems used in this paper is the same as that in (Davenport et al. 1994). In summary, the problems have 50 variables with a domain size of 10 each and randomly generated atmost constraints where $N = 3$, $|Var| = 5$ and $|Val| = 5$. Problem groups with 400, 405, 410, ..., up to 500 atmost constraints were generated. For each problem group, ten CSPs were generated. In order to test the reliability of the tunneling algorithm (i.e. how often it misses solutions), only soluble CSPs were chosen for comparison.

We tested 2ST against the MCHC and the GENET Stable1-Sideway model (which was referred to as GENET1 in (Davenport et al. 1994)) on this set of problems[8]. Because of the stochastic nature of these algorithms, we ran each algorithm 10 times for each problem. The limit for each run was set to 10,000 iterations (1 iteration = 1 GENET cycle) for GENET Stable1-Sideway model and for 2ST. The equivalent number of iterations for MCHC is 500,000 since MCHC examines only one variable per iteration. If a solution is found within the given number of iterations then this is considered as a successful run. For 2ST, the cost of the constraints and the penalty amount were all set to 1 and all the violated constraints were penalized in a local minimum. Figure 5 illustrates the percentage of successful runs for these algorithms, which measures their relative reliability. Both 2ST and the GENET Stable1-Sideway model clearly out-perform MCHC. For tightly constrained CSPs, 2ST out-performs the GENET Stable1-Sideway model — see the right-hand end of the chart where the percentage of successful runs declines faster for GENET than for 2ST.

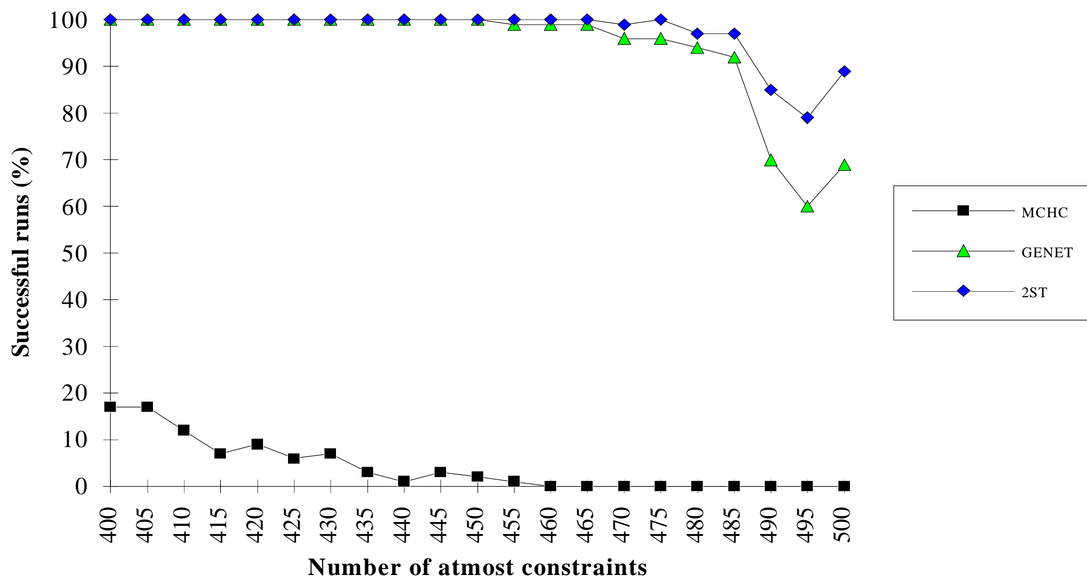

Figure 5. MCHC, GENET Stable1-Sideway, and 2ST on random general CSPs.

10.2 Hard Graph Coloring Problems

In this experiment, we tested 2ST on the four graph coloring problems from (Johnson et al. 1991) which have been suggested to be hard problems. We compare the results from 2ST with those from the GENET Stable1-Sideway model and GSAT, as they were reported in (Davenport et al. 1994). We counted a repair ("flip") for 2ST, each time the value of a variable

[8] 1ST is the same as GENET1 in CSPs.

was changed. The medians for the number of repairs, time and iterations are calculated over 10 runs for each problem. Table 1 presents these results, which show that 2ST out-performs both GENET and GSAT in terms of number of repairs and time.

Nodes	Colors	Median Number of Repairs			Median Time[9]			Median number of iterations	
		GSAT	GENET	2ST	GSAT	GENET	2ST	GENET	2ST
125	17	65,197,415	1,626,861	498,463	8.0h	2.6h	1.08h	524,517	131,385
125	18	65,969	7,011	4,949	30s	23s	17.6s	1,243	578
250	15	2,839	580	536	5s	4.2s	1.4s	15	12
250	29	7,429,308	571,748	338,571	1.8h	1.1h	1.1h	63,767	34,614

Table 1. GSAT, GENET, and 2ST on hard graph coloring problems.

10.3 Random Binary PCSPs

In order to examine the suitability of tunneling for Partial CSPs, we generated a set of random problems with soft binary constraints and assignment costs. The control parameters of the experiment are the *density* and *tightness* of the binary constraints. The density is defined as the probability of a pair of variables being constrained and the tightness is the probability of two labels of two constrained variables being incompatible. The constraints were represented in matrix format (Tsang, 1993) and we employed GENET's scheme for representing them (i.e. one constraint for each pair of incompatible values). Random costs in the range of 1 to 100 were assigned to the labels and in the range of 1,000 to 10,000 to the constraints. This basically means that the primary goal is to satisfy all the constraints, and the secondary goal is to choose low cost values (which is a reasonable thing to do in many scheduling problems). The objective function for all the problems is the combined cost of selected labels and violated constraints in the solution as given by (Eq. 9).

10.3.1 Design of Experiment

The randomly generated problems have 10 variables with domain size of 10. We varied the constraint density from 0.1 to 0.9 by increments of 0.1. For each density level, we varied tightness from 0.1 to 0.9 by increments of 0.1. This resulted in 81 distinct density-tightness configurations. Ten PCSPs were generated for each configuration, aggregating to a total of 810 problems. We solved all the problems to optimality with a Branch and Bound algorithm similar to Partial B&B presented by (Freuder & Wallace, 1992), appropriately modified to accommodate label and constraint costs. We tested 1ST and 2ST on the problems generated. Because of the stochastic nature of the tunneling algorithms, we ran each algorithm 10 times for each problem. The stopping criterion was set to 10,000 iterations. The FCR heuristic used for the selection of terms to be penalized and the penalty amount was specified by (Eq. 12).

10.3.2 Evaluation of Results

The main observation in this experiment is find out how close tunneling algorithms can get to optimal solutions in binary PCSPs. The performance of the tunneling algorithms are measured

[9] The experiments on the graph coloring problems were conducted on a Sun Microsystems Sparc Classic Workstation with a 2ST implementation in C++. The same settings were used as in (Davenport et al. 1994).

by the *relative solution error* also mentioned as percentage excess, which is defined by the following formula:

$$Error(\%) = \frac{g^{*} - g_{opt}}{g_{opt}} \times 100 \qquad \text{(Eq. 14)}$$

where g^{*} is the cost of the solution returned by the algorithm within a limit of iterations and g_{opt} the cost of the optimal solution. Analysis of variance (ANOVA) (Mendenhall & Sincich, 1992) was conducted on the 1ST and 2ST results. The dependent variable is the solution error and the three factors involved are:

- Algorithm: 1ST, 2ST (2 levels)
- Density: 0.1, 0.2,..., 0.9 (9 levels)
- Tightness: 0.1, 0.2,..., 0.9 (9 levels)

The main objective of the analysis is to evaluate 1ST and 2ST on the basis of error and also to determine interactions between the factors involved in the experiment. Table 2 shows the results from the ANOVA test.

Source of Variation	Degrees of Freedom	F	p-value
Algorithm	1	6.74	0.0094
Density	8	1.15	0.3269
Tightness	8	4.04	0.0001
Algorithm*Density	8	1.08	0.3701
Algorithm*Tightness	8	3.78	0.0002
Density*Tightness	64	2.35	0.0001
Algorithm*Density*Tightness	64	1.72	0.0003
error	16199	-	-

Table 2. Results of ANOVA test with dependent variable the solution error.

As we can see from the above table the effect on the solution error due to the algorithms is statistically significant (p=0.0094). Another significant factor is the tightness of the constraints (p=0.0001) while the density of the constraints alone or combined with the algorithms are not statistically significant (p=0.3269 and p=0.3701, respectively). We compared further 1ST and 2ST on the basis of solution error and time using Tukey's method for multiple comparisons (Mendenhall & Sincich, 1992). The results presented in Table 3 show that the 1ST and 2ST means on both solution error and time are significantly different.

Mean Solution Error		Mean Time in CPU seconds[10]	
1ST	2ST	1ST	2ST
0.03292%	0.01322%	6.87 sec	2.96 sec

Table 3. Performance comparison between 1ST and 2ST on PCSPs.

In fact, 2ST failed to find the optimal solution in only 51 out of 8100 runs while 1ST failed 208 times for the same number of runs. In those cases, near-optimal solutions were discovered by both algorithms. Figure 6 graphically illustrates the problems in which 1ST and 2ST failed. It shows that 1ST faced difficulties in tight PCSPs while 2ST in problems around the phase

[10] The experiments in random binary PCSPs were conducted on a Sun Microsystems Sparc Classic with 1ST, 2ST and B&B implemented in C++.

transition region (Smith, 1994). The overall performance of 2ST is significantly better than that of 1ST (note the change of scales in the z-axis in the timing and error).

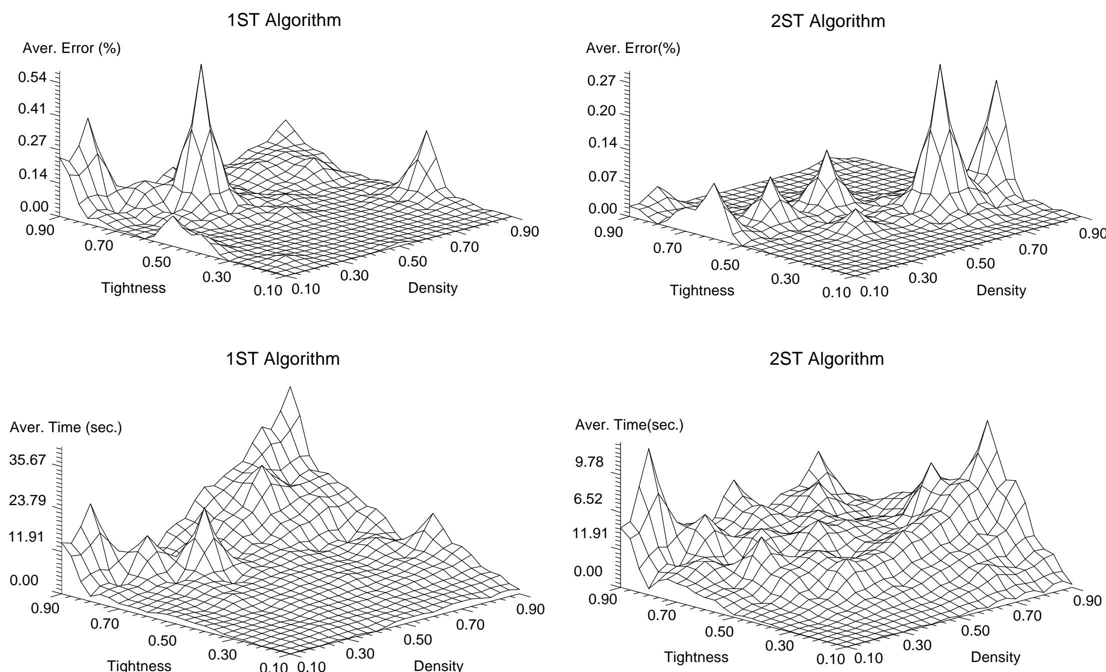

Figure 6. The surfaces[11] of solution error and time for 1ST and 2ST.

Note that the size of the problems tested is limited by the combinatorial explosion problem faced by the Partial B&B (Wallace & Freuder, 1993), which takes many hours to solve problems with 20 variables even when it provided with initial bounds generated by 2ST. Such results give justification for using stochastic methods such as the tunneling algorithm for PCSPs.

10.4 The Radio Link Frequency Assignment Problem — A Real Life PCSP

Randomly generated problems with known properties are useful for evaluating algorithm performance and therefore helping to guide the development of algorithms. However, we would also like to know how well the tunneling algorithms will perform in large CSOPs and PCSPs in real life applications.

We have tested the tunneling algorithms on the *Radio Link Frequency Assignment Problem*, which was abstracted from the real life application of assigning frequencies (values) to radio links (variables). Eleven instances of RLFAPs, which involve various optimization criteria, were made publicly available by the French Centre d'Electronique l'Armament (RLFAP, 1994). Two different types of binary constraints may be involved in these RLFAPs:

- The absolute difference between two frequencies must be greater than a given number k (i.e. for two frequencies X and Y, $|X - Y| > k$);
- The absolute difference between two frequencies must be exactly equal to a given number k (i.e. for two frequencies X and Y, $|X - Y| = k$).

[11] Inverse distance interpolation used.

Along with the problems publicized, the best solutions known so far (some of which were known to be optimal) were given. These solutions were found by tree-search algorithms, graph coloring algorithms and other methods. We applied both 1ST and 2ST to all the published RLFAPs instances. In the following sections, we shall first describe the optimization criteria of the problems; then we shall present our results.

10.4.1 Optimization Criteria

The problems involve both hard and soft constraints. If all the constraints can be satisfied then either:

- (C1) the solution which assigns the fewest number of different values to the variables,
- (C2) or the solution which largest assigned value is minimal

is preferred. For insoluble problems, *violation costs* are defined for the constraints. Furthermore, for some insoluble problems, default values are defined for the variables. If any of the default values is not used in the solution returned, then a predetermined *mobility* cost applies.

10.4.2 Problem Modeling

A. PCSPs

Modeling RLFAPs as PCSPs is straightforward. The primary constraint costs are set equal to the given violation costs. The primary label costs are all set to 0 for the default labels for the variables and to the mobility cost for all the other labels for each variable.

B. CSOPs

The two optimization criteria (C1) and (C2) are difficult to be modeled since both of them are non-linear — the cost of choosing a label depends on the values that other variables are taking in the current network state. To encourage the network to move towards the optimal solutions, these costs must be reflected in the primary label costs. We shall first explain how to model (C1) in our objective function.

The label cost of the label $l = <x, v>$ in the network state S is defined as follows:

$$l_v(S) = NVar - NVar_v^{'} \qquad \text{(Eq. 15)}$$

where $NVar$ the total number of variables in the problem and $NVar_v'$ the number of variables that are assigned the value v in state S. The tunneling cost corresponding to (Eq. 15) is defined as follows:

$$l_v(S) = NVar - NVar_v^{'} + penalty_v \qquad \text{(Eq. 16)}$$

where $penalty_v$ is defined for each possible value in the union of all the domains rather than separately for each individual label.

Firstly, the hard constraints of all the problems (PCSPs and CSOPs) were assigned high violation costs. Therefore, soft constraints and label costs are effectively ignored until the hard constraints are satisfied. In a local minimum, if hard constraints are violated, we penalize them all. The penalty amount used is defined in (Eq. 13).; i.e. the minimal penalty to enable the network to change state is applied to the terms selected for penalizing.

If all the hard constraints are satisfied then the FCR heuristic considers the label costs defined in (Eq. 15). In the selection, we exclude values not used by any of the variables because they cannot affect the cost of the local minimum once they are not used. Since a value may be used by many variables, in order to reduce the computational effort, we use (Eq. 12) for these penalty amounts.

The second criterion (C2) can be modeled in a similar way, but since criterion (C2) is only involved in one of the RLFAPs which solved as a CSP, we shall not elaborate it here. The results on running the 1ST and 2ST algorithms on the RLFAPs will be shown in the following sections[12] .

10.4.3 Problems 1,2,3,11 - Soluble Problems modeled as CSOPs

These problems have been modeled as CSOPs. The objective is to find the solution which minimizes the criterion (C1) defined above. We ran 1ST and 2ST ten times on each problem with the stopping criterion set to 20,000 iterations. The results obtained are consistent with the best solutions known so far. In Problem 2 the tunneling variants frequently found the publicized optimal solution. Problem 11 was initially thought to be insoluble and therefore was treated as a PCSP. It later turned out that the problem was soluble. Therefore, we proceeded further by treating problem 11 as a CSOP subject to criterion (C1). Table 4 summarizes the results obtained by running 1ST and 2ST on these four problems. It is worth emphasizing that the large number of variables and values involved in all of these problems prevents any known complete search algorithm from succeeding in solving them in reasonable time.

Problem	Number of Variables	Number of Constraints	Cost of Best Solution Known	Algorithm	Median Cost	Min Cost	Max Cost	Aver. Time (CPU sec.)	Aver. Iterations
1	916	5,548	16	1ST	18	16	48	123	655
				2ST	20	20	26	514	2,636
2	200	1,235	14 (optimal)	1ST	14	14	16	5	120
				2ST	14	14	18	21	486
3	400	2,760	16	1ST	16	16	18	181	1,964
				2ST	17	16	20	293	3111
11	680	4,103	Solution violates 0 constraints	1ST	41	32	Not solved	512	3,677
				2ST	43	30	Not solved	360	2,575

Table 4. RLFAP - Problems 1, 2, 3, and 11.

10.4.4 Problems 4,5 - Soluble Problems modeled as CSPs

Problem 4 requires optimization with respect to criterion (C1) and some of the variables in it are given default values. The optimization criterion for Problem 5 is (C2). Both 1ST and 2ST failed to solve these problems when they were treated as CSOPs. We believe that this failure was caused by the irregularity of the cost surface due to the costs and penalties added to the objective function from the optimization criteria which prohibited sideways moves. This does indicate that there is room for improvement for the tunneling algorithms. However, in ignoring the optimization criteria and attempting to solve these two problems as CSPs, 1ST and 2ST found solutions in all their ten runs within 20,000 iterations. Moreover, all the solutions happened to be optimal with respect to the corresponding criteria. Detailed results are presented in Table 5.

[12] The experiments on RLFAPs were conducted on DEC Alpha machines with 1ST and 2ST written in C++.

Problem	Number of Variables	Number of Constraints	Cost of Best Solution Known	Algorithm	Median Cost	Min Cost	Max Cost	Aver. Time (CPU sec.)	Aver. Iterations
4	680	3,967	46 (optimal)	1ST	46	46	46	5	58
				2ST	46	46	46	6	72
5	400	2,598	792 (optimal)	1ST	792	792	792	115	1,181
				2ST	792	792	792	155	1,628

Table 5. RLFAP - Problems 4 and 5.

10.4.5 Problems 6,7,8 - Insoluble Problems modeled as PCSPs

These problems have no known solutions which satisfy all the constraints. So they were treated as PCSPs. We solved the problems using the FCR heuristic and penalty amounts given by (Eq. 13). The stopping criterion was set to 20,000 iterations. Each problem was run ten times by the tunneling algorithm and the results are presented in Table 6. In contrast to the previous problems, 2ST found better solutions than 1ST. They both found far better solutions than the best known solutions so far in Problems 7 and 8.

Problem	Number of Variables	Number of Constraints	Cost of Best Solution Known	Algorithm	Median Cost	Min Cost	Max Cost	Aver. Time (CPU sec.)	Aver. Iterations
6	200	1,322	6,787	1ST	5,514	4,492	6,842	77	1,613
				2ST	4,661	3,702	5,253	402	8,332
7	400	2,865	2,545,752	1ST	63,162	47,104	1,063,987	1,001	9,621
				2ST	49,977	41,705	1,092,497	985	9,427
8	916	5,744	1,772	1ST	396	354	438	1,505	7,135
				2ST	384	317	427	1,867	8,776

Table 6. RLFAP - Problems 6, 7, and 8.

10.4.6 Problems 9,10 - PCSPs involving assignment costs

The problems 9 and 10 are PCSPs in which default values are provided for some of the variables. A number of variables must take these values while others may be assigned alternative values at certain costs (mobility costs). The sizes of these mobility costs are comparable to constraint violation costs. We modeled mobility costs in the way that is described in Section 10.4.2. We ran 1ST and 2ST five times on each problem up to 20,000 iterations with penalty amounts given by (Eq. 12). We have to admit here that the quality of solutions was much inferior for less generous penalty amounts like those determined by (Eq. 13). Table 7 summarizes the results of these experiments. Results by tunneling algorithm in Problem 9 were better than the best results found so far, but they were less impressive in Problem 10.

Problem	Number of Variables	Number of Constraints	Cost of Best Solution Known	Algorithm	Median Cost	Min Cost	Max Cost	Aver. Time (CPU sec.)	Aver. Iterations
9	680	4103	22,136	1ST	15,760	15,716	15,832	577	5211
				2ST	15,755	15,721	15,889	1081	9725
10	680	4103	21,552	1ST	31,518	31,517	31,521	576	5135
				2ST	31,520	31,517	31,618	332	2985

Table 7. RLFAP - Problems 9 and 10.

10.4.7 Evaluation of Results on RLFAP

In almost every instance of the publicized RLFAP (except Problem 10), the tunneling algorithms managed to find at least as good solution as the best solutions known so far. The running times required for these problems, which involved from 200 up to 916 variables, varied from a few seconds for the small CSOPs to less than an hour for the large PCSPs. Given the sizes and difficulty of the problems, we believe the times required by the tunneling algorithms were reasonable. Unfortunately, no information was given about the times required to find the best solutions publicized. In all the runs, very good solutions were available within a very short period of time. This makes the tunneling algorithms suitable for (time) resource bounded applications where early good solutions are welcome and further optimization is preferred whenever time allows. It is worth pointing out that the tunneling algorithm is a general algorithm. Only minor adaptations were required for it to be applied to the RLFAP, yet results obtained by it were comparable to, if not better than, the best known results so far.

10.5 The Traveling Salesman Problem

A classic problem in combinatorial optimization is the Traveling Salesman Problem (TSP). A challenge for us was to find out whether the tunneling algorithm was applicable to problems other than PCSPs. The TSP is well suited for tunneling for two reasons:

- Local search procedures based on 2-Opt heuristic (Lin & Kernighan, 1973; Papadimitriou & Steiglitz, 1982; Lawler et al. 1985; Aarts & Korst, 1989) alone are capable of finding near-optimal solutions.
- The objective function of TSP is appropriate for tunneling because it is given by a sum of terms. These terms can be modified by tunneling algorithm to guide the local search procedure out of local minima and towards better configurations.

We implemented a local search based on the 2-Opt heuristic as the local search procedure of the tunneling algorithm. At each iteration of local search, the best 2-Opt exchange was performed after considering all the possible moves (i.e. full neighborhood). The problems considered are Euclidean TSPs given by a matrix $D = [d_{ij}]$ where the element d_{ij} is the Euclidean distance between the cities i and j. A tunneling penalty matrix $P = [p_{ij}]$ is defined where the element p_{ij} is the penalty added to the element d_{ij} of the distance matrix D. Additionally, a frequency count is kept for each possible edge, this is used by the FCR heuristic in the selection of tunneling directions. The cost of the edge e_{ij} that connects cities i and j is a function of the tour:

$$e_{ij}(Tour) = \begin{cases} d_{ij} & , e_{ij} \in Tour \\ 0 & , e_{ij} \notin Tour \end{cases} \quad \text{(Eq. 17)}$$

Following Garfinkel & Nemhauser (1972), the objective function of the TSP is defined as follows:

$$g(Tour) = \sum_i \sum_j e_{ij}(Tour) \quad \text{(Eq. 18)}$$

To form the tunneling function, we consider the tunneling version of (Eq. 17), which is as follows:

$$e_{ij}^T(Tour) = \begin{cases} d_{ij} + p_{ij} & , e_{ij} \in Tour \\ 0 & , e_{ij} \notin Tour \end{cases} \quad \text{(Eq. 19)}$$

The tunneling function for TSP is defined as follows:

$$g^T(Tour) = \sum_i \sum_j e_{ij}^T(Tour) \qquad \text{(Eq. 20)}$$

In the TSP, the terms of the objective function to be modified are the edge lengths. Once the terms for modification have been identified, the basic mechanism of tunneling is the same as the one for PCSPs. 1ST and 2ST use 2-Opt as the basic local search procedure. Through the tunneling function, 2-Opt sees the edge lengths as given by (Eq. 19). Due to penalty terms, edges may be considered longer than they really are. In a local minimum (i.e. there is no 2-change that improves the tour), an edge is selected using the FCR heuristic presented in Section 9.1 and an amount equal to the primary distance is added to its penalty. In tie situations, only one edge was penalized. The new penalties force 2-Opt to consider moves which were not considered before, thus escaping from the local minimum.

Since the TSP is not modeled as a PCSP, the pseudocode of modified 1ST (modified for tackling TSPs) is given in Figure 7.

```
PROCEDURE 1ST-TSP(g, g^T, Tour*)
BEGIN
        k ← 0;
        Tour_k ← arbitrary legal tour;
        best_cost_so_far ← g(Tour_k);
        best_tour_so_far ← Tour_k;
        REPEAT
                REPEAT
                        2-Opt(g^T, Tour_k, Tour_k+1);
                        IF (best_cost_so_far > g(Tour_k+1) THEN
                                BEGIN
                                        best_cost_so_far ← g(Tour_k+1);
                                        best_tour_so_far ← Tour_k+1;
                                END
                        k ← k+1;
                UNTIL (Local_Minimum)
                INCREASE_PENALTIES(Tour_k);
        UNTIL (Stopping_Criterion);
        Tour* ←  best_tour_so_far;
END
```

Figure 7. The 1ST algorithm for the Traveling Salesman Problem.

The basic difference between this algorithm, which we call 1ST-TSP, and the standard 1ST for PCSPs is in the use of 2-Opt instead of the GENERATE procedure. Similar modifications were made for 2ST, resulting in the algorithm 2ST-TSP which will not be elaborated here.

10.5.1 Experimentation with TSPs

In this section, we shall present results for 1ST-TSP and 2ST-TSP on a set of ten well-studied TSP problems where the optimal solutions are known. These problems are from TSPLIB (Reinelt, 1991), a library of TSP problems. The stopping criterion was set to 100,000 iterations. Ten runs were made for each problem. Table 8 presents the results of our runs.

Problem Identity	No. Cities	Average Excess (%) (Eq. 14)		Average Time (CPU seconds[13])		Optimal Runs (out of 10)		Aver. No. Iterations	
		1ST	2ST	1ST	2ST	1ST	2ST	1ST	2ST
eil51	51	0.00	0.09	4.94	21.8	10	6	3006	13383
eil76	76	0.00	0.11	20.3	43.5	10	5	5629	11952
eil101	100	0.00	0.10	82.3	41.9	10	5	12537	6380
kroA100	100	0.09	0.09	135.8	305.6	2	4	21185	47341
kroC100	101	0.00	0.27	196.0	145.8	10	2	30782	23036
lin105	105	0.16	0.00	73.6	17.4	4	10	10577	2522
kroA150	150	0.09	0.37	779.2	1111.8	2	1	41598	53670
kroA200	200	0.43	0.65	2176.1	1062.9	0	0	60468	30370
lin318	318	1.05	1.42	4617.7	5403.5	0	0	50412	57679
pr439	439	0.89	1.10	9866.4	8171.8	0	0	55745	47224

Table 8. Table of results for the Traveling Salesman Problem.

The algorithms in all the runs improved the solutions found by 2-Opt alone (i.e. the first local minima). If enough time was given the algorithms were able to find the optimal solution in most problems. Between the algorithms, 1ST-TSP had a better overall performance than 2ST-TSP. Figure 8 shows the fluctuation of the cost during the 500 iterations of 1ST-TSP on a problem with 51 cities. As can be seen from the figure, after the initial steep drop in cost, the algorithm visited solutions with near-optimal costs. Eventually, it obtained the optimal cost.

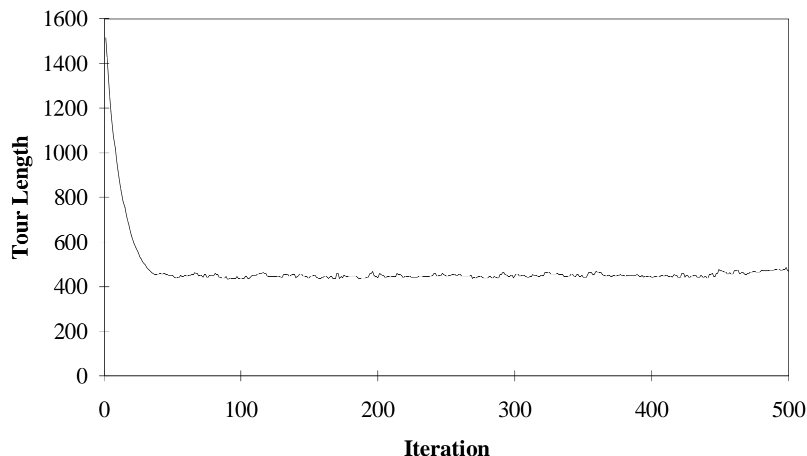

Figure 8. The cost of the solutions generated in the first 500 iterations of 1ST for a Traveling Salesman Problem with 51 cities.

Note that both 1ST-TSP and 2ST-TSP alter the cost function only after the first local minimum is reached and therefore in the worst case their performances are at least as good as 2-Opt. This is not necessarily the case for other methods which are also built upon 2-Opt, for example tabu search and simulated annealing. Furthermore in tunneling, no parameter other than the upper limit of iterations is necessary to be specified (unlike some other methods such as genetic algorithms). It must be emphasized that the experimentation with TSPs presented is subject to improvement since alternative edge selection heuristics (other than FCR) can be devised and approximate 2-Opt based local search procedures might be used. Discussion of such improvements will be left for another occasion.

[13] TSP experiments were conducted on DEC Alpha Machines with 1ST-TSP and 2ST-TSP written in DEC C++.

11. Future Work

Further study will focus on analyzing when the tunneling algorithm will perform better than other methods, why and how it can be improved. In order to improve the efficiency and effectiveness of the algorithm, the following issues will be look at:

- Penalty Amounts
- Selection Procedure
- Termination Criterion

The efficiency of the algorithm is strongly related to the first two decisions. Applying large penalties in general will speed up the algorithm at the risk of burying optimal solutions among inferior solutions. Low penalties have the opposite effect of slowing down the algorithm but reducing such risks. The FCR heuristic presented in this paper has worked well for the problems studied so far. However, different selection procedures may be more efficient for the same or other kinds of problems. Finally, for the termination criterion, more sophisticated schemes may be explored.

12. Summary and Conclusions

Many real life problems can be formulated as partial constraint satisfaction problems (PCSPs), which involves the assignment of values to variables satisfying a set of constraints and optimizing according to a specified objective function whenever possible.

GENET is a connectionist approach to constraint satisfaction. In this paper, we have introduced the *tunneling algorithm*, which is extended from GENET to include optimization, for solving PCSPs. We have also demonstrated the generality of the tunneling algorithm by applying it to a combinatorial optimization problem, namely the TSP.

Results so far indicate that the tunneling algorithm is applicable to a variety of problems. It has been shown to be robust in satisfiability problems: it managed to solve the tested problems in more cases than some of the best algorithms published so far (namely the MCHC, GSAT and the GENET Stable1-Sideway Model). It has also been demonstrated to be effective on optimization problems: in applying it to a number of publicly available RLFAPs and TSPs, it has consistently been able to find solutions with quality which is as good as, and sometimes much better than, that in the best solutions found so far. Moreover, we found it relatively easy to adapt the algorithm to different applications.

Despite the incompleteness of the tunneling algorithm, it had no difficulties in producing optimal solutions repeatedly in a high proportion of the problems tested within reasonable time. Therefore, this algorithm should be a powerful tool for tackling problems which backtrack-based algorithms find intractable.

To summarize, the tunneling algorithm has been shown to be a robust and effective algorithm for constraint satisfaction and optimization problems.

Acknowledgments

We would like to thank the Centre d'Electronique l'Armament for making available the RLFAP instances which initiated our development of the tunneling algorithm and The British Telecom Research Laboratories for making available a number of real instances of network scheduling problems which lead us to the development of the 2ST scheme. Andrew Davenport provided us with his implementation of the GENET Stable1-Sideway model and its results in tackling the graph coloring problems. Both Andrew Davenport and James Borrett commented on an earlier

version of this paper. The Computing Service of the University of Essex gave us access to the DEC Alpha machines which helped us to carry out the experiments. This project and Chris Voudouris are funded by the EPSRC grant (GR/H75275).

References

Aarts, E., & Korst, J. (1989). *Simulated Annealing and Boltzmann Machines*. New York: John Wiley & Sons.

Barto A. G., Sutton R. S., & Anderson C. W. (1983). Neurolike adaptive elements that can solve difficult learning problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13, 834-846.

Davenport A., Tsang E., Wang C. J., & Zhu K. (1994). GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of AAAI-94*, 325-330.

Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., & Graf, T. (1988). Applications of CHIP to industrial and engineering problems. In *First International Conference on Industrial and Engineering Applications of AI and Expert Systems*.

Freuder, E. C., & Wallace, R. J. (1992). Partial constraint satisfaction. *Artificial Intelligence*, 58, 21-70.

Garfinkel, R. S., & Nemhauser, G. L. (1972). *Integer Programming*. New York: John Wiley and Sons.

Glover, F. (1989). Tabu search Part I. *Operations Research Society of America(ORSA) Journal on Computing*, Vol. 1, 109-206.

Glover, F. (1990). Tabu search Part II. *Operations Research Society of America(ORSA) Journal on Computing*, Vol. 2, 4-32.

Johnson, D., Aragon, C., McGeoch, L., & Schevon, C. (1991). Optimization by simulated annealing: an experimental evaluation, part II, graph coloring and number partitioning. *Operations Research,* 39(3), 378-406.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science,* 220, 671-680.

Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G., & Shmoys D. B. (Eds.) (1985). *The Traveling Salesman Problem: A guided tour in combinatorial optimization*. Chichester: John Wiley & Sons.

Levy, A.V., & Montalvo, A. (1985). The tunneling algorithm for the global minimization of functions. *SIAM Journal on Scientific and Statistical Computing*, 6, No. 1, 15-29.

Lin, S., & Kernighan, B. W. (1973). An effective heuristic algorithm for the Traveling salesman problem. *Operations Research*, 21, 498-516.

Mendenhall, W., & Sincich, T. (1992). *Statistics for engineering and the sciences - 3rd ed.* New York: Maxwell Macmillan International.

Minton S., Johnston, M.D., Philips, A. B., & Laird, P. (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58, 161-205.

Morris, P. (1993). The breakout method for escaping from local minima. In *Proceedings of AAAI-93*, 40-45.

Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice-Hall.

Reinelt, G. (1991). A Traveling Salesman Problem Library. *Operations Research Society of America(ORSA) Journal on Computing*, Vol. 3, 376-384.

RLFAP (1994). The RLFAP problems are available from *listserver@saturne.cert.fr*. To get more information and the problems send the command "get csp-list CELAR.blurb" to the above listserver. Problems used in our tests were obtained on March 1994.

Selman, B., & Kautz, H. A. (1993). Domain dependent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 290-295.

Selman, B., Levesque, H. J., & Mitchell, D. G. (1992). A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92*, 440-446.

Selman B., Kautz H. A., & Cohen B. (1994). Noise strategies for improving local search. In *Proceedings of AAAI-94*, 337-343.

Smith, B., M. (1994). Phase transition and the mushy region in constraint satisfaction problems. In *Proceedings of 11th European Conference on Artificial Intelligence*, 263-267.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*. London: Academic Press.

Wallace R. J., & Freuder E. C. (1993). Conjunctive Width Heuristics for Maximal Constraint Satisfaction. In *Proceedings of AAAI-93*, 762-768.

Wang, C. J., & Tsang, E. (1991). Solving constraint satisfaction problems using neural-networks. In *Proceedings of IEE Second International Conference on Artificial Neural Networks*, 295-299.

Wang, C. J., & Tsang, E. (1992). A cascadable VLSI design for GENET. In I*nternational Workshop on VLSI for Neural Networks and Artificial Intelligence*, Oxford.

Whittle P. (1971). *Optimization under Constraints: Theory and Applications of Nonlinear Programming*. London: John Wiley & Sons.

Zhigljavsky, A. A. (1991). *Theory of Global Random Search*. London: Kluwer Academic Publishers.