

A Comparison of Two Methods for Solving 0-1 Integer Programs Using a General Purpose Simulated Annealing Algorithm

David Abramson

*School of Computing and Information Technology, Griffith University,
Kessels Rd., Nathan, Qld 4111. Australia
davida@cit.gu.edu.au*

Henry Dang

*Dept. of Computer Systems Engineering, Royal Melbourne Institute of Technology,
P.O. Box 2476V, Vic 3001, Australia.*

and

Mohan Krishnamoorthy

*Division of Mathematics and Statistics,
CSIRO, Private Bag 10, Rosebank MDC, Clayton, Vic 3168, Australia.
mohan@mel.dms.csiro.au*

Abstract

0-1 problems are often difficult to solve. Although special purpose algorithms (exact as well as heuristic) exist for solving particular problem classes or problem instances, there are few general purpose algorithms for solving practical-sized instances of 0-1 problems. This paper deals with a general purpose heuristic algorithm for 0-1 problems. In this paper we compare two methods based on simulated annealing for solving general 0-1 integer programming problems. The two methods differ in the scheme used for neighbourhood transitions in the simulated annealing framework. We compare the performance of the two methods on the set partitioning problem.

1. Introduction

1.1 General Introduction

0-1 integer programming problems are a very important class of integer programming problems. Some of the important problems belonging to this class are knapsack, assignment, matching, covering, packing, partitioning, facility location, travelling salesman, fixed charge network flow, and so on. Many practical problems can be formulated as instances of the above problems. Apart from the above direct applications, 0-1 variables are often used in modelling instances where objective functions are discontinuous or when disjunctive constraints are present (for example in scheduling problems). 0-1 problems are often difficult to solve. Although special purpose algorithms (both exact as well as heuristic) exist for solving particular problem classes or problem instances, there are few general purpose algorithms for solving practical-sized instances of 0-1 problems. This paper deals with a general purpose heuristic algorithm for 0-1 problems. For a comprehensive treatment of 0-1 problems, refer to Nemhauser and Wolsey [31].

0-1 integer programming problems can either be solved exactly (using methods like branch and bound) or with the help of special-purpose heuristic algorithms. Generally, heuristic methods for 0-1 programs tend to be problem specific. In other words, a good heuristic for one 0-1 problem may not necessarily be good for another problem.

Connolly [15] gives a general purpose algorithm based on simulated annealing (SA) for solving general 0-1 problems. In this paper, we extend the results of Connolly [15] and enhance his method. We also compare two methods of performing the general purpose SA on our test problem, the set partitioning problem (SPP).

A brief introduction to the general principles of SA is provided in Section 1.2. The general methodology for a general purpose SA for solving 0-1 problems is provided in Section 1.3. The two neighbourhood transition approaches are described in detail in Section 2. In Section 4, we introduce enhancements to the basic SA, some of which are specific to the SPP, our test problem (which is described in Section 3). Section 5 deals with specific algorithm implementation details. Detailed computational results are presented in Section 6. We also provide directions for further research in Section 7.

Even though the paper describes an SA implementation to solve the SPP, the results are generally applicable to other 0-1 problems. Further, the structural properties of the SPP which are exploited to reduce the problem size are only included for the sake of completeness, and do not affect the general conclusions. Although the paper describes the implementation of the general purpose SA algorithm on the classical formulation of the SPP, it derives its generality from the fact that it can handle additional constraints that are not problem-specific. This feature of a general purpose algorithm is discussed in greater detail in Connolly [15].

1.2 Simulated Annealing

Simulated Annealing (SA) is a powerful technique for solving combinatorial optimisation problems (COP). SA is random in nature, general in characteristic and simple to implement (see, for example, Collins et al, [14]). These characteristics make SA suitable for solving a wide range of COP's (see van Laarhoven, et al [36]).

A general COP will have many solution states, X , each with a cost, $C(X)$. SA searches the solution space of a COP based on random perturbations of the current state, X and a probability of acceptance of the new state, X' . This probability can be computed by evaluating the expression, $e^{-\Delta C/T}$, where $\Delta C = C(X) - C(X')$ and T is a control parameter (the temperature). The acceptance criteria is such that all transitions which decrease the cost are accepted. Transitions which cause an increase in cost are accepted if the value of the above expression is greater than a randomly chosen number in the interval $[0,1)$. Thus, given a current state (solution configuration) and its neighbourhood solution space, a new configuration within the neighbourhood structure is selected. The change in cost is computed and the acceptance criteria is used to determine which configuration will be the starting state for the next perturbation. In the next section, we consider the application of an SA, similar to the one described above, for solving general 0-1 problems.

1.3 SA for General 0-1 Integer Programming Problems

Consider the 0-1 integer programming problem

$$\begin{aligned} \min \quad & C(X) = \sum_{j \in J} C_j x_j \\ \text{Subject to} \quad & AX \otimes b, \\ & x_j \in \{0,1\}. \end{aligned}$$

Where, $\otimes$ is either $\leq$, $=$, $\geq$ or a mixture of these,

$$X = x_j, j \in J,$$

A is an $(m \times n)$ matrix,

b is an m -vector.

The general purpose SA algorithm for solving the above problem starts with an initial configuration, its associated cost, and a given starting temperature. At each subsequent temperature, the algorithm iterates by perturbing the current configuration an appropriate number of times, called the Markov chain length. Perturbation is simulated by toggling the value of a randomly chosen binary variable x_j from 0 to 1 or vice versa. It is possible that, by changing the value of chosen variable x_j we may obtain an infeasible solution. We may, at this stage, accept the infeasible solution with a resulting penalty, or restore feasibility. (We deal with this issue in greater detail in Section 2). The new configuration is accepted if the change in cost is negative. Otherwise, the acceptance probability is used to determine whether the uphill move is accepted. The mechanism for transforming the current solution into one of its neighbourhood is called a *Markovian transformation*, in which, the current configuration only depends on the immediately previous one. The temperature is then decreased according to one of a number of cooling schedules (Collins et al, [14]). The process is continued until there is no change in cost between a number of consecutive Markov chains.

The basic form of the SA algorithm may be represented as follows

```

Generate initial configuration call  $X$ 
Compute cost of initial configuration  $C(X)$ 
Compute initial temperature  $T_0$ 
Set  $T = T_0$ 
  While not termination do begin
    Repeat Markov-Chain-Length times begin
       $i = \text{random}(0, n - 1)$ 
      Set  $X' = \text{flip}(X, i)$ 
      IF using force feasible approach, THEN
        restore feasibility of  $X'$ 
       $\Delta C = C(X') - C(X)$ 
      IF ( $\Delta C \leq 0$ ) or ( $e^{-\Delta C/T} > \text{unif\_rand}(0,1)$ ), THEN
        Set  $X = X'$ 
      end
     $T = \text{Cool}(T)$ 
  end
end

```

where $\text{flip}(X, i)$ will flip the value of the i 'th element of X and return the resulting vector.
 $\text{random}(a, b)$ returns a random uniformly distributed integer between a and b
 $\text{unif_rand}(a, b)$ returns a random uniformly distributed number between a and b
 $\text{Cool}(T)$ returns a value for the temperature after applying a cooling operation to T .
 In the experiments reported in this paper we use a reheating scheme in which the temperature is reduced and then raised to escape from local minima

2. Two Neighbourhood Transition Approaches

SA takes no account of the problem constraints when it performs the Markovian transformation. Thus, any general purpose algorithm should include mechanisms for incorporating problem-specific constraints while performing neighbourhood transitions. There are two main techniques that can be used to incorporate these constraints. The first only accepts feasible transitions, where only feasible solutions are considered. The other admits infeasible transitions with a resulting penalty.

In this paper we employ efficient implementations of both of these procedures to our test problem, the Set Partitioning Problem (SPP) and compare their performance.

2.1 Penalise Infeasibility

In this approach, the constraints of the problem are relaxed and incorporated into the objective function by using a penalty function. Constraints can be relaxed in part or completely. The penalty function can be linear or non linear, uniform or non uniform, continuous or discrete, subject to the condition that, when the new cost function (augmented by the penalty) is minimised, the original objective is minimised and the constraints are satisfied.

With this approach, the search can proceed through infeasible regions in the hope that a feasible, and lower cost region will be found by the SA. Thus, any *point* in the search space (even infeasible ones) can be admitted as a solution. This provides flexibility while deriving the appropriate transition operator of the SA. Details of such an implementation can be found in Abramson, et al [3, 5]. This approach has the added ability of assigning priorities to constraints. The result is that constraints can be categorised into groups of hard and soft. Another attraction of this approach is that it can be used to solve problems in which there exists only a very small set of feasible solutions. Details of an application on one such problem can be found in Abramson and Dang [2].

Despite its attractiveness, particularly for problems with difficult constraints, this neighbourhood transition approach has a number of disadvantages. The new objective function can be highly non-linear even if the original objective function is linear. The search can stop at a local minimum which may not even be inside the feasible region. Also, as the problem size increases, the above approach may cause the length of the search to increase dramatically, because the SA may randomly wander in a potentially large search space. Finally, a new cost function is required every time a new problem is encountered. This limits the generality of such an SA algorithm.

2.2 Restore Feasibility

In this approach, only feasible solutions are accepted. As a result, once a transition, called the *primary transition*, is performed, a sequence of transitions are required to restore feasibility. This approach differs from the scheme which penalises infeasible solutions because the search can only proceed through the feasible region. Therefore, the approach carries a computational overhead between each transition, because it is necessary to restore feasibility.

This approach can be applied to a problem in which feasibility can be easily restored. It has the advantage that all partial solutions are feasible. Moreover, there is no need for deriving a new cost function for each new problem encountered. So, the core algorithm can be applied to any general 0-1 IP problem directly.

Connolly [15] proposed a general purpose SA for 0-1 IP problems in which all neighbourhood transitions are based on feasibility restoration. In his algorithm, feasibility restoration is based on a heuristic. The effectiveness of this heuristic depends on the proportion of feasible points in the search space.

Initially, a variable index is randomly chosen. If its value is 0, it is changed (or, *flipped*) to 1, or vice versa. If feasibility of the solution is undisturbed, then, there is no necessity to invoke the feasibility restoration procedure, and the algorithm continues. If feasibility is disturbed, the heuristic performs a sequence of resulting moves (variable value flips) on selected variables in an attempt to restore feasibility. The selection criteria is based on S_j , the *help-score* for each variable (or, *column*). The help score for a column depends on how much it helps the current infeasible constraints (or, *rows*) and on how *critical* these rows are. The criticality of a row i is denoted by K_i and is

measured as follows:

$$K_i = \frac{INF_i}{HELP_i}$$

$$\text{where } INF_i = \begin{cases} \max(0, LHS_i - RHS_i) & , \text{ if } constraint \leq \\ |RHS_i - LHS_i| & , \text{ if } constraint = \\ \max(0, RHS_i - LHS_i) & , \text{ if } constraint \geq \end{cases}$$

$$HELP_i = \sum_{j \in H_i} |a_{ij}|$$

where H_i is the set of columns j whose variable values can be flipped to help reduce infeasibility in row i
 LHS_i is the current value of the left hand side of row i
 RHS_i is the right hand side of row i

From the above, it is clear that as K_i approaches 0, the criticality of row i decreases. Values of K_i close to 1, but with $K_i < 1$ indicate that the current infeasibility of row i can only be restored by flipping the variable values of most of the *helpful* columns associated with row i (columns belonging to H_i). $K_i > 1$, indicates that restoring feasibility to row i has become impossible, since even if the remaining 'helpful' columns were flipped, the row would remain infeasible. In this case, a backtracking move is required. The help-score for each column j is then computed as

$$S_j = \sum_{i \in I} K_i \times \min(|a_{ij}|, INF_i)$$

where, I is the set of infeasible rows.

The column with the maximum S_j is then chosen as the next column whose variable value will be flipped in an attempt to restore feasibility. If a tie occurs, then the column which will cause the least new infeasibility is chosen. If this test fails to separate the candidate columns, then the objective function values are used to resolve the issue.

If feasibility is restored as a result of the above choice, then the algorithm continues by evaluating the cost of the transition and either accepting or rejecting it. If not, the value of K_i , $HELP_i$, INF_i and S_j are recomputed and another column is chosen to be flipped in an attempt to restore feasibility. If, at any stage in the above sequence, $K_i > 1$ for any i , the most recent column whose variable value was flipped is restored to its original value and removed out of all H_i that it is included in. The values of K_i , $HELP_i$, INF_i and S_j are recomputed and the process continues.

Furthermore, if, as a result of a primary transition, the number of variables flipped in the bid to restore feasibility reaches a pre-defined limit, all variable values that were flipped in the sequence are reset to their original value. No further attempt is made to restore feasibility. The algorithm continues with a new choice of primary transition.

The speed of the feasibility restoration procedure is proportional to the size of the problem. Hence, problem size is a major factor affecting the speed of convergence of SA using the above approach. For problems with enormous search space, the time consumed between each transition may be large. To overcome this problem, we add on

a few search space reduction techniques to reduce the problem size as the algorithm proceeds. These techniques speed up the feasibility restoration procedure. Some of the techniques are general for 0-1 IP problems, but others can only be applied to our test example, the set partitioning problem.

3. The Set Partitioning Problem

Let $M = \{1, \dots, m\}$ and $N = \{1, \dots, n\}$ denote the set of rows and columns, respectively, in an $(m \times n)$ matrix, A . Further, assume that the elements of the matrix A take values of either 0 or 1. Let a_{ij} denote the (i, j) th element of A . Associated with each column j , of the matrix A is a cost C_j .

Then, the set partitioning problem (SPP) on matrix A can be formulated as follows :

$$\begin{aligned}
 P_{SPP} : \quad & \min \sum_{j=1}^n C_j x_j \\
 \text{subject to :} \quad & \sum_{j=1}^n a_{ij} x_j = 1, \quad \forall i \in M \\
 & x_j \in \{0, 1\}.
 \end{aligned}$$

Most applications of SPP arise in the transportation and scheduling area, scheduling of trucks, tanker routing, switching circuit design, assembly line balancing, capital equipment decisions, location of off shore drilling platforms, scheduling of ships, scheduling of airline fleets and airline crew scheduling (see Gershkoff [19], Hoffman and Padberg [21], Marsten et al [27], Marsten and Shepardson [28], McCloskey and Hanssman [29]). Detailed surveys on the SPP, including its applications, can be found in Balas and Padberg [6, 7], Christofides, et al [13], Garfinkel [17] and in Garfinkel and Nemhauser [18].

A popular application of the SPP is the airline crew scheduling problem (CSP). The goal is to minimise crew costs while satisfying constraints on work/duty content. Given an airline's flight schedule (arrivals and departures at all destinations, including aircraft type), *pairings* are initially formed. Each pairing is a sequence of flight *legs* which begin and end at a base location. A pairing is also known as a *tour of duty*. Each pairing has a cost associated with it. Pairings are normally governed by operational policies, aviation regulations, union rules, and so on. Hence, given an exhaustive set of all legal (feasible) pairings, the problem is to find the best (least cost) set of pairings such that each flight leg is covered by exactly one pairing. Each row of the matrix $A = [a_{ij}]$ represents a flight leg. Each column of A denotes a pairing in which a value of 1 denotes a flight that is included. The value of the 0-1 variable x_j indicates whether a pairing is selected in the optimal solution or not. CSPs are normally very large problems. For example, in excess of a billion feasible pairings could easily result for problems involving just 800 flight legs. Hence, the search space can be enormous. More details on CSPs can be found in Hoffman and Padberg [21].

4. Search Space Reduction Techniques

For problems with enormous solution space, the SA guided search for a good solution could be time consuming, due to the random nature of SA. Moreover, if the constraints are tight (as in the SPP), the feasibility restoration procedure, if used, could also

consume large amount of time. In this section, we illustrate techniques to reduce the search space as SA proceeds. This also enables in speeding up the feasibility restoration procedure.

4.1 Using LP Reduced Costs

Consider the 0-1 Integer Programming Problem

$$P_{0-1} : Z_{0-1} = \min CX \quad (1)$$

$$\text{subject to : } AX \leq b \quad (2)$$

$$x_j \in \{0,1\} \quad (3)$$

Here C is an n -vector, A is an $(m \times n)$ matrix and b is an m -vector. The problem P_{0-1} has an optimal solution value, Z_{0-1}

Consider the problem LP_{0-1}

$$LP_{0-1} : Z_{LP} = \min CX$$

$$\text{subject to : } AX \leq b$$

$$X \leq 1$$

$$X \geq 0$$

LP_{0-1} is an LP relaxation of P_{0-1} . It follows that $Z_{LP} \leq Z_{0-1}$; the optimal solution to LP_{0-1} forms a lower bound to the solution of P_{0-1} . The lower bound can be used to measure the quality of the upper bounds, particularly in problems like the SPP, in which the LP lower bounds are normally very tight. Let Z_{UB} be the upper bound obtained through the SA heuristic; $Z_{LP} \leq Z_{0-1} \leq Z_{UB}$. The gap between Z_{LP} and Z_{UB} can, therefore, be used to gauge the effectiveness of the upper bounding algorithm. Second, Z_{LP} can be used in conjunction with Z_{UB} to *prune* the search space.

Consider the problem LP_{0-1} and let X^{LP} be the optimal solution vector. Let $X_B^{LP} \subset X^{LP}$ be the set of basic variables and let $X_J^{LP} \subset X^{LP}$ be the set of non basic variables in the LP optimal solution X^{LP} . For any non basic variable $x_k \in X_J^{LP}$, we can define the reduced cost, C'_k , as the amount by which the objective function coefficient of x_k must be improved (decreased) before x_k could assume a positive value in the optimal solution. Another interpretation of C'_k , the reduced cost of a non basic variable $x_k \in X_J^{LP}$ is that it is the amount by which the value of Z_{LP} will change (increase, in a minimisation problem) if we increase the value of x_k by 1 from its current value of 0 (while all the other non basic variables remain equal to zero), thereby introducing it into the basis. By definition, the reduced cost of basic variables is zero.

In 0-1 problems, the reduced cost information provides us with a measure to evaluate an increase by 1 (the upper bound) in the activity level of non basic variables from their value of 0 in X^{LP} . Thus the introduction of a non basic variable, $x_k \in X_J^{LP}$ in the basis has the effect of an increase in the lower bound from Z_{LP} to $Z_{LP} + C'_k$. For all $x_k \in X_J^{LP}$, if $Z_{LP} + C'_k > Z_{UB}$, then x_k can never be part of any optimal solution to P_{0-1} and hence, can be eliminated from the problem.

In this manner, all non basic variables in X^{LP} can be considered, in turn, as candidates for elimination. This process often results in significant reductions in problem size. We will illustrate this in Section 5.

Hoffman and Padberg [21] use reduced costs in conjunction with a *reduce-cost-fixing* procedure to permanently fix variables. This procedure is described in Crowder et al [16] and Padberg and Rinaldi [34]. Jaumard, et al [22] use the LP reduced costs (as described above) as well as other penalty schemes for SPP instances arising from real world traffic assignment systems. LP preprocessing has been used previously in conjunction with SA. Cattrysse [11] and Cattrysse, et al [12] have applied a variable reduction procedure that fixes some variables, x_{ij} to 1 when solving the generalised assignment problem (GAP). This results in problem size reduction. Their procedure is based on the addition of valid inequalities (or facets) of the knapsack polytope. The procedure begins by solving an LP relaxation of the GAP. Based on the solution, the formulation is strengthened by the addition of a valid inequality for every knapsack constraint that is violated. The new formulation is resolved to generate more facets and a better bound. The procedure is repeated till no further inequalities can be found. At the end of this procedure, all variables with $x_{ij} = 1$ are removed from the problem and the formulation is adjusted. The reduced problem is solved using an SA procedure. The benefits of incorporating the above preprocessing for the GAP are increased speed and the ability to solve larger problems.

In our approach, we use the LP reduced costs in combination with the current best cost obtained by the SA to dynamically prune the search space. Thus, as the SA improves the best known cost, more variables can be eliminated.

4.2 Using Structural Properties of The Problem

Apart from the LP pruning scheme, the structural properties of the problem can be used to prune the search space further. This also limits the search space for moves resulting from primary transitions in the feasibility restoration procedure and hence, speeds it up. These are common-sense reduction procedures, and have been used in algorithms for the SPP (for example, in Hoffman and Padberg [21]).

Let J be the set of rows associated with non-zero entries in column j . Let K be the set of rows associated with non-zero entries in column k . If j is known to be in the solution, any column k , such that $(K \cap J) \neq \emptyset$, can not play any part in any optimal solution. This feature can be incorporated into the problem by maintaining a count of the number of columns remaining in each row. Once, a count reaches 1, the column that is associated with that row must be in the solution. We call this column an *indispensable* column. Any other column that has at least one common row with an indispensable column must have its variable value fixed to zero and removed out of the search space permanently. We call such columns *superfluous*.

If the number of remaining variables in the problem reduces to a certain critical level (150 in our experiments), then we change the restore feasibility search policy to one that completely enumerates the remaining search space. Under such a policy, if a column (called the *primary column*) has been flipped from 1 to 0 at any stage of the SA and if it is not possible to restore feasibility after the whole search space has been explored, this column must be indispensable. As a result, we can fix to 0, the variable values of the columns that have at least one common row with the primary column. Such columns can be removed permanently out of the search space as they are superfluous. Similarly, if a column (called the *primary column*) has been flipped from 0 to 1 at any stage of the SA and if it is not possible to restore feasibility after the whole search space has been explored, this column must be superfluous. As a result, we can fix the value of this column to 0 permanently.

In either of the above cases, in order to speed up the feasibility restoration procedure, we use smart data structures, in combination with the features of the SPP, to guide our search of the solution space for alternate feasible solutions. Assume column j is flipped from 0 to 1 and let J be the set of rows associated with non-zero entries in j . Let S be the set of columns in the search space. Let p be any column whose variable value is currently 0 and let P be the set of rows associated with non-zero entries in p . Let q be any column whose variable value is currently 1 and let Q be the set of rows associated with non-zero entries in q . Let $Free$ be the set of columns whose variable values have not been altered since the primary transition. Every column $p \in S$, $(P \cap J) \neq \emptyset$, $p \in Free$, must be temporarily removed out of the search space, and x_p must be temporarily fixed to 0. Every column $q \in S$, $(Q \cap J) \neq \emptyset$, $q \in Free$, must also be temporarily removed out of the search space, and x_q must be temporarily set to 0.

These fixed values are only relative to the current value of the primary column, j . If, as a result of the above pruning (relative to the primary column), it is impossible to restore feasibility, we drop out of the restoration procedure and declare the result of the feasibility restoration as negative. Alternatively, we add in succession, other columns in an attempt to restore feasibility, while continually pruning the search space as a result of such additions. If it is possible to restore feasibility as a result of a transition involving the primary column, the transition is accepted. All temporary changes have to be undone and the columns involved have to be re-included in the search space before the SA can proceed.

Further we can embed problem reduction process called redundancy preprocessing, proposed by Hoffman and Padberg [21], throughout the SA. In this paper, we use this procedure to reduce problem size only at the beginning of the SA. For further details of this process, refer to Hoffman and Padberg [21]

5. Implementation Details

As stated in Section 2, we implemented two different SA-based algorithms for solving our test problem, the SPP: one that restores feasibility and another which penalises infeasibility after every transformation. In this section, we will differentiate between the different implementation details relating to the two algorithms only where necessary. We denote the SA that penalises infeasible moves as PISA. The SA that restores feasibility is denoted as RFSA.

5.1 Data structure

Sparse matrix representation is employed to denote the characteristics of the data structure. Each column is uniquely defined by its own feature such as cost, number of rows, LP reduced cost, etc. Columns are arranged in the order of increasing reduced cost to speed up search space pruning. A similar structure is applied to row indexing wherein, each row is uniquely denoted by the number of columns and a pointer to a list of non-zero column elements that are associated with it. The reason for maintaining the list of row indexed accessing is to speed up the search space pruning and feasibility restoration procedure.

In RFSA, there are two additional sets of columns to be maintained: forbidden set and sub-forbidden set. The former is a static set which contains columns that have their variable values permanently fixed. The latter is a dynamic set which contains columns that have their variable values fixed temporarily (as described in Section 4.2).

5.2 Cooling Schedule

A cooling schedule consists of (i) The starting temperature. (ii) The length of the Markov Chain. (iii) The cooling function. (iv) The process termination condition

- (i) We computed our starting temperature based on White [37], who proposed that the starting temperature should be the temperature at which the expected cost is just within one standard deviation of the mean cost. In practice this amounts to computing the standard deviation of the cost for a fixed number of transitions, all of which are accepted. The initial temperature is then set to this value.
- (ii) Since a feasible solution exists after each move in RFSA, we assume that a constant Markov Chain length of 1 is sufficient. In PISA, we use a constant Markov chain length of 10.
- (iii) The process is terminated if one of the following cases happen: (a) ten consecutive Markov chains provides no change in the solution; (b) a specified time limit is reached; (c) all columns have been fixed or removed; (d) If the number of remaining columns is reduced to a certain level (we used a limit of 10 in this study), at which point, we perform a complete enumeration to find the solution.
- (iv) We used a fix cooling rate of 0.9. This is based on Kirkpatrick et al [24, 25] who proposed that the temperature decrement rule should be controlled by a constant cooling factor which is small but close to 1. This rule is used by, for example, Johnson, et al [23], Bonomi and Lutton [8, 9, 10], Leong and Liu [26], Morgenstern and Sharpiro [30], Sechen and Sangiovanni-Vincentelli [35].

We use a temperature reset procedure which increases the temperature if a local minimum is detected. A local minimum is reached if two consecutive Markov chains (our default value) provides no change in the solution, or if there is a fluctuation around a constant value. The latter condition is used to detect cycling. The temperature is reset according to

$$T_{new} = P \times C(x^b) + Q$$

where T_{new} = new reset temperature when a local minimum is detected
 x^b = the current best solution
 $C(x^b)$ = the cost of the current best solution
 P and Q = constants

This reset temperature is a function of the current best cost and a precomputed base temperature (Q). The current best cost value is used to limit the maximum increase in cost which is allowed. Thus, as the quality of the solution improves, the search is prohibited from directly retracing its steps. The value of P is used to scale cost values into the temperature domain. In the annealing of physical systems, a phase changes may occur as the system is cooled, and the corresponding temperature is referred to as the specific heat of the substance. In our implementation, Q is set to the specific heat. The justification for choosing this temperature is that by reheating the system just past that point, it is possible to re-anneal through the phase change. Further details are discussed in Abramson, et al [3, 4].

Temperature resets within the framework of SA have been previously applied to several other problems. Connolly [15] applies a geometric reheating scheme for general purpose SA. Osman and Christofides [32] apply a non-monotonic cooling schedule for the capacitated clustering problem (CCP). The authors also provide a classification of cooling schedules in the literature along with a comparison of their performance on the

CCP. In Hasan and Osman [20], the authors show the sensitivity of parameter selection in reheating schemes, as applied to an SA algorithm for the weighted maximal planar layout problem. The non-monotonic cooling schedule has also been applied to the vehicle routing problem in Osman[1993].

6. Computational Results and Discussion

6.1 Comparison of PISA and RFSA

In this section we present performance results of the PISA and RFSA algorithms. The data sets for the test problems are taken from Hoffman and Padberg [21], and these represent a substantial set of real world air crew scheduling problems. The problem characteristics are summarised in Table 1. Table 2 presents the results gained by using PISA, whilst Table 3 shows the results of the same test problems using the RFSA algorithm. Tables 2 and 3 only report the results of the first 37 data sets. The remaining 5 are discussed in Section 6.2.

Each problem originally consists of a number of columns (variables) and rows (constraints). Using the redundancy preprocessing methods of Hoffman and Padberg [21] it is possible to reduce the problem size, and the number of rows and columns after preprocessing are presented in the fifth column of Table 1. The number of non zero elements and problem density is shown in columns 6 and 7 of the table.

The two SA algorithms are run on each test problem 6 times, each with a different random starting solution. All tests were performed on a Sun Sparc Station 2. In the case of RFSA, the starting solution is forced to be feasible. Tables 2 and 3 show the best and average relative percentage deviations (RPD) over all 6 runs and for all the test problems. Here RPD is defined as the relative percentage different between the SA cost and the known optimal cost (as reported in Hoffman and Padberg [21]), and is given by:

$$RPD = \frac{SA\ Cost}{Known\ Optimum\ Cost} \times 100\%$$

Tables 2 and 3 also present average solution times. These times do not include the preprocessing time and the LP solution time. On average, both preprocessing and solving the initial LP each take about 5% of the PISA solution time. As the SA proceeds it attempts to reduce the problem size further by using pruning techniques mentioned in Section 4. The remaining columns after the SA are presented in each of the tables

From Table 2, it can be seen that in many cases the SA algorithm using PISA returns an optimal solution for all six independent runs, e.g. the first 29 problems with the exception of Test number 8. For Test number 8, most of the runs result in an optimal solution, as observed by zero deviation of the best cost and a small deviation of the average cost from the optimal cost. It appears that this deviation was caused by limiting the run time. In almost all of the first 29 cases the number of columns remaining after the SA search is quite small, indicating that the pruning techniques have been successful. This aspect of the algorithm is discussed in more detail in Abramson, et al [2]. The last 8 problems are significantly more difficult, mostly because they are much larger. Also, the SA is unable to form an upper bound which is tight enough to reduce the size of the search space. Consequently, the resulting solutions are of poor quality and in some cases are actually infeasible. This is a characteristic of PISA that the final solution may be infeasible. Whilst some of the early test problems are solved quite quickly by PISA, the more difficult problems require many minutes. It is worth comparing these times to those achieved using a specialised solution technique, like the algorithm for crew scheduling instances of the SPP, proposed by Hoffman and Padberg [21]. Their paper reports solution times in the order of and below seconds.

Consequently, our scheme should not be considered as an efficient algorithm for large scale SPPs. It is, however, general in nature and can incorporate additional constraints.

At first inspection of Table 3, most of the problems are solved in less than one minute. Also in most cases, the algorithm results in an exhaustive search (variables values are all fixed), thus, the final solution is known to be optimal. It can be seen that RFSA has managed to solve some problems that were not solved by PISA (like problems 30, 31, 33, 35 and 37). An anomaly in the case of test data 5 can be seen, where RFSA has taken longer than PISA to solve. This is due to the nature of the constraints in this problem. Because it is difficult to find a feasible solution, RFSA spends more time in making moves than PISA. Because PISA is able to move through infeasible regions it is able to converge on the solution faster.

Figure 1 and 2 show the relative performance of PISA and RFSA. Figure 1 shows the relative average run times of PISA and RFSA. This can be expressed as $\frac{\text{Average Time to Solve (PISA)}}{\text{Average Time to Solve (RFSA)}}$. We have only reported this ratio for problems which were solved for all 6 runs, thus it only covers tests 1 through 29. It is clear from this graph that RFSA is much faster in comparison to PISA in all but one problem.

Figure 2, shows the relative performance difference in cost between PISA and RFSA. This is computed as $\frac{SA \text{ Average RPD (PISA)}}{SA \text{ Average RPD (RFSA)}}$. The results for tests 32 and 34 are not reported because RFSA was unable to find even one feasible solution. This is because the data sets are highly constrained. Further, PISA is unable to find feasible solutions for some of the problems, and this is reflected by the inclusion of a penalty cost in the final solution cost.

It appears that the RFSA algorithm out performs the PISA technique in almost all cases.

Test No	Problem Name	Rows	Columns	Number of (Rows x Columns) After Pre-Processing	Number of Non Zeroes	Density
1	nw41	17	197	17 x 177	740	22.1%
2	nw32	19	294	18 x 251	1357	24.3%
3	nw40	19	404	19 x 336	2069	27.0%
4	nw08	24	434	21 x 352	2332	22.4%
5	nw15	31	467	29 x 405	2830	19.5%
6	nw21	25	577	25 x 421	3591	24.9%
7	nw22	23	619	23 x 521	3399	23.9%
8	nw12	27	626	25 x 451	3380	20.0%
9	nw39	25	677	25 x 565	4494	26.6%
10	nw20	22	685	22 x 536	3722	24.7%
11	nw23	19	711	18 x 423	3350	24.8%
12	nw37	19	770	19 x 639	3778	25.8%
13	nw26	23	771	21 x 468	4215	23.8%
14	nw10	24	853	21 x 655	4336	21.2%
15	nw34	20	899	20 x 718	5045	28.1%
16	nw43	18	1072	17 x 982	4859	25.2%
17	nw42	23	1079	23 x 795	6533	26.3%
18	nw28	18	1210	18 x 582	8553	39.3%
19	nw25	20	1217	20 x 844	7341	30.2%
20	nw38	23	1220	21 x 723	9071	32.3%
21	nw27	22	1355	22 x 817	9395	31.5%
22	nw24	19	1366	19 x 926	8617	33.2%
23	nw35	23	1709	23 x 1191	10494	26.7%
24	nw36	20	1783	20 x 1244	13160	36.9%
25	nw29	18	2540	18 x 2034	14193	31.0%
26	nw30	26	2653	26 x 1877	20436	29.6%
27	nw31	26	2662	26 x 1728	19977	28.9%
28	nw19	40	2879	32 x 2134	25193	21.9%
29	nw33	23	3068	23 x 2308	21704	30.8%
30	nw09	40	3103	38 x 2301	20111	16.2%
31	nw07	36	5172	34 x 3105	41187	22.1%
32	aa02	531	5198	360 x 3846	36359	1.3%
33	nw06	50	6774	38 x 5956	61555	18.2%
34	aa06	646	7292	492 x 5898	51728	1.1%
35	kl01	55	7479	47 x 5915	56242	13.7%
36	us02	100	13635	45 x 5766	192716	14.1%
37	nw13	51	16043	50 x 10903	104541	12.8%
38	us04	163	28016	99 x 4205	297538	6.5%
39	kl02	71	36699	69 x 16542	212536	8.2%
40	nw03	59	43749	53 x 38956	363939	14.1%
41	nw01	135	51975	135 x 49903	410894	5.9%
42	nw17	61	118607	61 x 78186	1010039	14.0%

Table 1 : Test Problems

Test No	Number of Columns After SA	PISA Best RPD%	PISA Average RPD%	PISA Average Time (Secs)
1	17	100	100	27.6
2	25	100	100	33.9
3	18	100	100	32.3
4	20	100	100	324.9
5	30	100	100	68.9
6	16	100	100	66.9
7	23	100	100	344.8
8	27	100	109	1276.6
9	20	100	100	68.4
10	34	100	100	118.7
11	57	100	100	312.6
12	19	100	100	30.8
13	22	100	100	78
14	22	100	100	971
15	21	100	100	64.3
16	21	100	100	509.2
17	39	100	100	152.8
18	18	100	100	18.1
19	21	100	100	155.2
20	25	100	100	19.3
21	15	100	100	69
22	57	100	100	77.1
23	18	100	100	150.5
24	108	100	100	328.1
25	38	100	100	306
26	36	100	100	111
27	36	100	100	155.9
28	21	100	100	608.3
29	39	100	100	1404.4
30	734	102	103	5064.3
31	20	100	Infeasible	7987.1
32	3846	Infeasible	Infeasible	9182.5
33	81	100	Infeasible	12226.8
34	5892	Infeasible	Infeasible	10499.8
35	5915	Infeasible	Infeasible	7546.4
36	5766	Infeasible	Infeasible	10379.9
37	10899	Infeasible	Infeasible	8623.1

Table 2 : Results Of Using Penalised Infeasibility Approach (PISA)

Test No	Number of Columns After SA	RFSA Best RPD%	RFSA Average RPD%	RFSA Average Time (Secs)
1	0	100	100	18.1
2	4	100	100	10.2
3	0	100	100	20.2
4	0	100	100	97.2
5	12	100	100	226.2
6	0	100	100	4.1
7	0	100	100	32.6
8	0	100	100	12.1
9	0	100	100	26.8
10	0	100	100	16.8
11	8	100	100	62.6
12	0	100	100	23
13	5	100	100	8.7
14	0	100	100	14.2
15	0	100	100	38.9
16	0	100	100	41.3
17	5	100	100	16.2
18	0	100	100	4.4
19	0	100	100	17.3
20	0	100	100	12.4
21	0	100	100	30
22	18	100	100	23.7
23	0	100	100	14.7
24	14	100	100	27.4
25	0	100	100	27.2
26	7	100	100	30.6
27	7	100	100	37.8
28	0	100	100	42.5
29	0	100	100	62.3
30	0	100	100	311.7
31	0	100	100	616.9
32	N/A	Infeasible		
33	6	100	100	11662
34	N/A	Infeasible		
35	203	100	101	5293.5
36	4705	116	127	16402.1
37	30	100	100	17944.8

Table 3 : Results Of Restore Feasibility Approach (RFSA)

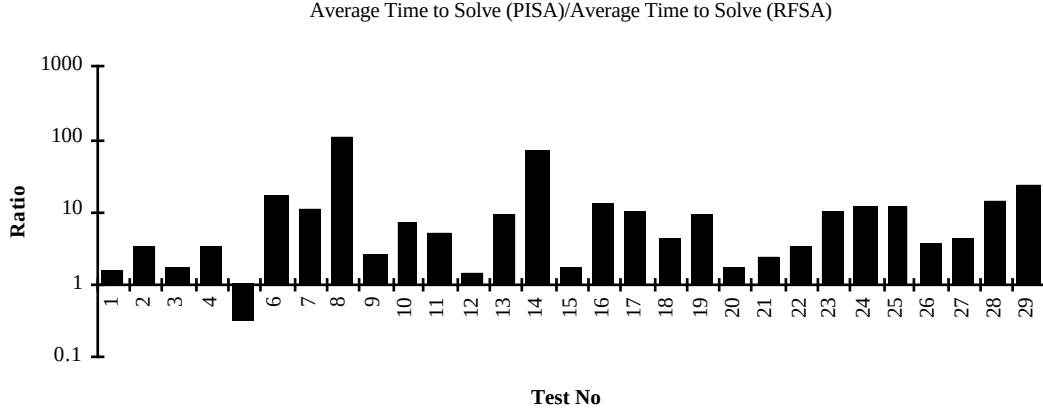


Figure 1 : Performance Comparison (RPD-Time To Solve)Between The Two Approaches

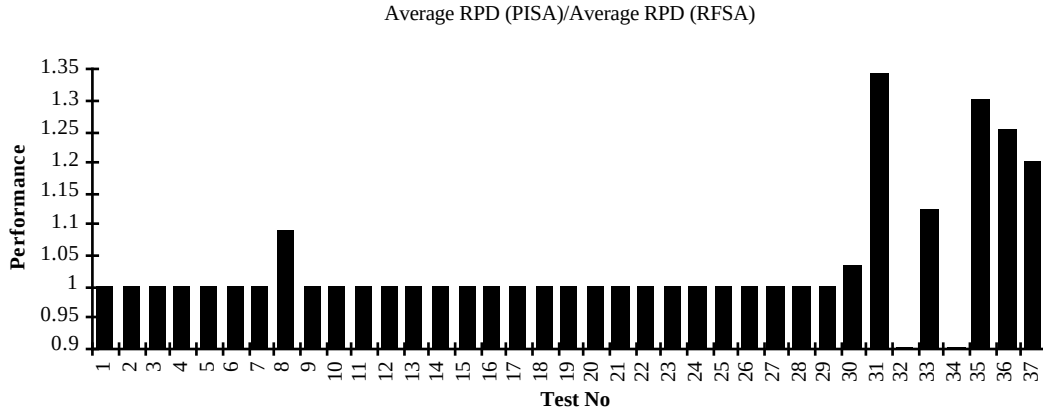


Figure 2 : Performance Comparison (Average RPD) Between The Two Approaches

6.2 Solving Large SPPs with RFSA

The experiments reported in the previous section have demonstrated that RFSA outperforms PISA for the SPP instances that we tested. However, we were interested in determining whether our general purpose SA is an effective solution technique for solving large SPPs, and thus we tried some larger data sets. These problems are taken from the data sets in Hoffman and Padberg [21] and are shown as tests 38 through 42 in Table 1. It can be seen that they contain from 28,000 to 118,000 variables, and have densities ranging from 6.5-14%. These problems were solved in Hoffman and Padberg [21] by a very efficient special purpose algorithm. Because of the superiority of RFSA over PISA, we only attempted RFSA on these larger problems.

On 4 of the 5 problems RFSA failed to generate an initial feasible solution. This appears to have occurred because the large number of variables and constraints means that there are too many potential combinations to be searched in order to establish feasibility. On the remaining test, the average cost was up to 6 times larger than the optimal cost. In these problems, it was not possible to use the LP reduced costs and the SA upper bound to reduce the search space. Hence, the SA algorithm failed.

Thus, from our experimental work it appears that SA is not a suitable method for solving very large SPPs. However, there are some possible extensions which were not explored in our work. First, the SA is unable to generate a feasible solution because the search space is so large, and the heuristic for establishing feasibility is incapable of finding even one feasible point. We did not experiment with other heuristics for creating an initial feasible solution, or for navigating the search space. For all of the smaller problems this did not represent a problem. Thus, more research needs to be conducted into suitable feasibility heuristics. Second, it may be possible to remove some of the more difficult constraints and incorporate them as penalties into the objective function, as done in PISA. This would allow the RFSA algorithm to make some progress whilst at the same time taking account of the complex constraints. Third, in Hoffman and Padberg [21], a simple heuristic is used to generate an upper bound for their search procedure. The heuristic uses the reduced costs and applies linear relaxations recursively until it generates an integer solution. For many of the problems this generates a sufficiently tight upper bound that the initial search space could be pruned dramatically before it is presented to the SA algorithm. This would appear to be an attractive path, and can be applied to SPPs regardless of the search algorithm. However, the thrust of this paper is an investigation of two SA approaches for solving general 0-1 problems. Therefore, we felt that a special-purpose algorithm that uses the above extensions to solve large SPPs, was beyond the scope of the study. It is, however, possible that such extensions would prove fruitful in developing an effective SA-based heuristic for large SPPs.

7. Conclusions and Further Research

In this paper we have presented a general purpose simulated annealing algorithm for solving 0-1 integer programming problems. The algorithm incorporates two significant extensions of the paper by Connolly [15]. The first is in the direct use of the LP solution and shadow prices to dynamically prune the SA search space. The second, is the interesting result that, except for a small set of difficult problems, restoring feasibility at each stage works better than using an approach that accepts infeasible transitions with a resulting penalty.

We have tested our algorithm on some real world air crew scheduling problems. Using standard pruning techniques, we are able to prune the entire search space for most of the problems and shrink the search space of many problems dramatically. This method allows us to generate solutions which are provably optimal in many cases. This latter result, which is unusual for most implementations of simulated annealing, has been commonly used in exact methods like branch and bound. Further work in producing some generic bounding schemes, like the reduced LP scheme, would be of significant value.

Even though the paper describes the SA implementation to solve real world SPP instances, the results are generally applicable to other 0-1 problems. The problem specific structural properties which are exploited to reduce problem size are only included for the sake of completeness, and do not affect the general conclusions. Furthermore, the algorithm derives its general purpose nature from the knowledge that it can also handle additional constraints that are not problem-specific. For example, the work done by Connolly [15] allows for arbitrary combinations of inequality constraints, in addition to the equality constraints required by the SPP formulation.

Although the RFSA approach performs better than the PISA approach, it fails for problems that are heavily constrained. On the basis of our experimental results, it appears that PISA is also incapable of finding feasible solutions for heavily constrained problems. Further, on small problems, PISA is much slower than RFSA. In order to implement an effective general SA algorithm for 0-1 IP problems, it may be possible to use a hybrid algorithm which switches between the two systems. Such an approach would need to detect 'hard' constraints, and introduce them into the objective function

(as in PISA), and solve the remaining problem using RFSA. Such detection could be performed during the RFSA initialisation phase whilst the algorithm is generating an initial feasible solution. The exact method for identifying the initial hard constraints is an open research issue.

As mentioned in Section 6, the times we have achieved for solving the SPP problem are an order of magnitude slower than those reported using a specialised algorithm like *branch-and-cut*. Thus, simulated annealing should not be considered an appropriate technique for solving the SPP problem where speed is the primary objective. Further, the experimental results have indicated that SA is unable to solve very large SPPs. However, the SA approach has the advantage that it is more generic and can be applied to a number of different problems with little modification. Moreover, additional constraints can also be handled easily. Techniques which can accelerate the simulated annealing approach include parallel computers and highly specialised processors (see Abramson [1, 2]). We are currently investigating a specialised attached processor which can be used for the algorithms proposed in this paper.

Acknowledgments

Henry (Mong Hung) Dang is supported by an Australian Postgraduate Research Award (Industry) and Computer Techniques P/L. Thanks go to Mr. John Shaw of Computer Techniques for his assistance in this work. We also express our appreciation to the anonymous referees for their helpful comments and hints on improving the original submission.

References

- [1] D.A. Abramson, A Very High Speed Architecture to Support Simulated Annealing, IEEE Computer, (1992).
- [2] D.A. Abramson and H. Dang, School timetables: A case study in simulated annealing, in: *Applied Simulated Annealing, Lecture Notes in Economics and Mathematical Systems*, ed, V.V. Rene Vidal, (1993), pp 103-124.
- [3] D.A. Abramson, H. Dang and M. Krisnamoorthy, Enhanced simulated annealing through linear programming preprocessing, Proceedings of the 12th Australian Society for Operations Research (ASOR) Conference, Adelaide (1993).
- [4] D.A. Abramson, H. Dang and M. Krisnamoorthy, Cooling schedules for simulated annealing based scheduling algorithms, Technical Report, School of Computing and Information Technology, Griffith University, Nathan , Qld 4111, Australia..
- [5] D.A. Abramson, H. Dang and M. Krisnamoorthy, Enhancing simulated annealing for 0-1 problems with linear programming preprocessing, Technical Report, School of Computing and Information Technology, Griffith University, Nathan , Qld 4111, Australia..
- [6] E. Balas and M.W. Padberg, Set partitioning, MSRR No. 351, Carnegie Mellon University (1974).
- [7] E. Balas and M.W. Padberg, Set partitioning: A survey, SIAM Review, 18(1976)710-760.
- [8] E. Bonomi and J.L. Lutton, The N-city travelling salesman problem: Statistical mechanics and the metropolis algorithm, SIAM Rev, 26(1984)551-568.
- [9] E. Bonomi and J.L. Lutton, The asymptotic behaviour of quadratic sum assignment problems: A statistical mechanics approach, European Journal of Operations Research, 18(1984).

- [10] E.Bonomi and J.L. Lutton, Simulated annealing algorithm for the minimum weighted perfect euclidean matching problem, *R.A.I.R.O Recherche Operationelle*, 20(1986)177-1282.
- [11] D.G. Cattrysse, Set partitioning approaches to combinatorial optimization problems, PhD Thesis, Catholique Universiteit Leuven, Dept Wertaigkunde, Appeling Industriel Beleid, Belgium (1990)
- [12] D.G. Cattrysse, D.G. and L.N. Van Wassenhove, A survey of algorithms for the generalized assignment problem, *European Journal of Operations Research*, 60(1992)260-272.
- [13] N. Christofides, et. al, *Combinatorial Optimization*, Wiley (1979).
- [14] N.E. Collins, R.W. Eglese and B.L. Golden, Simulated annealing: An annotated bibliography, *American Journal of Mathematical and Management Sciences*, 8, Nos 3 & 4, (1988)209-307.
- [15] D.T. Connolly, General purpose simulated annealing, *Journal of Operational Research*, 43(1992)495-505.
- [16] H. Crowder, E.L.Johnson and M.Padberg, Solving large scale zero-one linear programming problems, *Operations Research*, 31(1983)803-834.
- [17] R.S. Garfinkel, Optimal set covering: A survey, in A. Geoffrion (Ed.), *Perspectives on Optimization*, Addison-Wesley, Reading, MA (1972).
- [18] R.S. Garfinkel and G.L. Nemhauser, The set partitioning problem: Set covering with equality constraints, *Operations. Research*, 17(1969)848-856.
- [19] I. Gershkoff, Optimizing flight crew schedules, *Interfaces*, 19(1989)29-43.
- [20] M. Hasan and I.H. Osman, Local search strategies for the weighted maximal planar graph problem, *International Transactions in OR*, Vol 1, No.4(1994).
- [21] K.L. Hoffman and M. Padberg, Solving airline crew-scheduling problems by branch-and-cut, *Management Science*, 39(1993)657-682.
- [22] B. Jaumard, M. Prais and C.C.Ribeiro, Penalty computations for the set partitioning problem, *Annals of Operations Research* 43(1993)285-293.
- [23] D.S. Johnson, C.R. Aragon, L.A. McGeoch and C. Schevon, Optimization by simulated annealing: An experimental evaluation, *Workshop on Statistical Physics in Engineering and Biology*, Yorktown Heights (1986).
- [24] S. Kirkpatrick, C.D. Gelatt Jr and M.P. Vecchi, Optimization by simulated annealing, *IBM Research Report*, RC 9355. (1982)
- [25] S. Kirkpatrick, C.D. Gelatt Jr and M.P. Vecchi, Optimization by simulated annealing, *Science*, 220(1983)671-680.
- [26] H.W. Leong and C.L. Liu, Permutation channel routing, *Proceedings of the IEEE International Conference on Computer Design*, Port Chester (1985)579-584.
- [27] R.E. Marsten, M.R. Muller and C.L. Killion, Crew planning at Fly Tiger: A successful application of integer programming, *Management Science*, 25,(1979)1175-1183.
- [28] R.E. Marsten and R. Shepardson, Exact solution of crew scheduling problems using the set partitioning model: Recent successful applications, *Networks*, 11(1981)165-177.
- [29] J.F. McCloskey and F. Hanssman, An analysis of stewardess requirements and scheduling for a major airline, *Naval Research Logistics Quarterly*, 4(1957)183-192.

- [30] C.A. Morgenstern and H.D.Shapiro, Chromatic number approximation using simulated annealing, Technical Report CS86-1, Department of Computer Science, The University of Mexico, Albuquerque, (1986).
- [31] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*, Wiley (1988).
- [32] I.H. Osman and N. Christofides, Capacitated clustering by hybrid simulated annealing and tabu search, *International Transactions in OR*, 1(1994).
- [33] I.H. Osman, Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem, *Annals of Operations Research*, 41(1993)421-451.
- [34] M. Padberg and G. Rinaldi, A branch-and-cut algorithm for the solution of large scale travelling salesman problems, *SIAM Review*, 33(1990)1-41.
- [35] C. Sechen and A.L. Sangiovanni-Vincentelli, The Timber Wolf placement and routing package, *IEEE J. Solid State Circuits*, SC-20(1985)510-522.
- [36] P.J.M. van Laarhoven and E.H.L. Aarts, *Simulated Annealing: Theory and Applications*, Reidel, The Netherlands (1987).
- [37] S.R. White Concepts of scale in simulated annealing, *Proceedings. IEEE International Conference on Computer Design*, Port Chester, (1984)646-651.