A simulated annealing code for general integer linear programs★

David Abramson[a] and Marcus Randall[b]

[a]*School of Computer Science and Software Engineering, Department of Digital Systems, Monash University, Clayton, VIC 3168, Australia*

E-mail: davida@dgs.monash.edu.au

[b]*School of Environmental and Applied Science, Griffith University, QLD 4217, Australia*

E-mail: m.randall@eas.gu.edu.au

This paper explores the use of simulated annealing (SA) for solving arbitrary combinatorial optimisation problems. It reviews an existing code called GPSIMAN for solving 0–1 problems, and evaluates it against a commercial branch-and-bound code, OSL. The problems tested include travelling salesman, graph colouring, bin packing, quadratic assignment and generalised assignment. The paper then describes a technique for representing these problems using arbitrary integer variables, and shows how a general simulated annealing algorithm can also be applied. This new code, INTSA, outperforms GPSIMAN and OSL on almost all of the problems tested.

Keywords: simulated annealing, combinatorial optimisation, integer linear programming

AMS subject classification: 90C05, 90C10, 90C27

1. Introduction

For many years, the holy grail of operations research has been an efficient integer solver which can be applied to a wide range of problems with little modification. In this model, a user specifies a problem using some mathematical notation which is processed directly by the solver. One of the most general specification techniques uses a linear cost function and constraints, together with variables which are restricted to the values 0 and 1. There are a number of algorithms which attempt to find solutions for these 0–1 problems; however, their performance is highly variable.

★ The authors would like to acknowledge the assistance of Mohan Krishnamoorthy in this work. Marcus Randall is funded by a Griffith University Post Graduate Scholarship. The project is funded by the Australian Research Council.

In 1992, Connolly published a general purpose simulated annealing algorithm for solving arbitrary 0–1 linear integer programs [7]. Connolly applied the code (called GPSIMAN) to a number of problems, and compared the performance of the heuristic to a commercial integer programming package, called SCICONIC [7]. The code was tested on nine problems of varying size. In general, GPSIMAN performed quite well in finding the optimal solution to each of the problems. GPSIMAN is novel in that it takes standard 0–1 models and incorporates a general mechanism for restoring feasibility of the system after each move. GPSIMAN adopts a different approach to many other general SA codes such as Ingber's ASA (Adaptive Simulated Annealing) [10] and Johnson et al. [11,12]. These systems may be considered toolboxes in which there is a core SA engine that can determine an appropriate cooling schedule for a particular problem and the user provides a set of special purpose software modules. The modules are necessary to provide an interface to the particular problem being solved and include functions that

- calculate the objective cost;
- determine the state of the constraints;
- perform the neighbourhood transition.

GPSIMAN, on the other hand, accepts an algebraic problem description and hence requires no additional code. The system developed for this study is an enhancement of GPSIMAN and follows its approach of focusing on solving problems given an abstract algebraic representation. This method lends itself to very fast prototyping of a wide range of combinatorial optimisation problems without the development of new computer code every time a problem is to be solved. However, it is expected that general solvers will always execute more slowly than a tailored code.

We have evaluated GPSIMAN on a wider range of problems than in Connolly's paper, and have compared the results with a state-of-the-art commercial IP solver, namely IBM's OSL [9]. Our results indicate that GPSIMAN is not competitive when compared with OSL using the original data sets, and that on larger problems, both codes behave poorly. One of the problems of GPSIMAN stems from its generality. Arbitrary 0–1 problems tend to incorporate many more constraints than an equivalent version which uses general integer variables. The consequence of this is that GPSIMAN spends about 30% of its time restoring feasibility after each transition. Since many of the 0–1 constraints are present to preserve a binary encoding of the solution, restoring them is necessary for every change in a binary variable. However, using general integer variables instead of binary ones usually causes the cost function and constraints to become nonlinear, which can make the problem more difficult to solve.

In this paper, we discuss a solver which can accept problems formulated using arbitrary integer variables. The techniques used by the solver are novel because it is possible to detect the problem structure and use this information to guide the search,

effectively eliminating most encoding constraints. This is achieved both by widening the scope of transition operators available to general solvers and by broadening the algebraic notation used to express combinatorial optimisation problems. The new code (INTSA) that incorporates these features outperforms both GPSIMAN and OSL on almost all of the problems we have tested. The paper begins with an overview of Connolly's original GPSIMAN. We then introduce a number of classical problems written in both 0–1 form and also in our integer form. The algorithms necessary to solve the integer formulations are given, followed by some results.

2. General simulated annealing for 0–1 problems

It is possible to express many combinatorial optimisation problems in a form which uses binary integer variables. This approach has good expressive power, and has been used to represent a wide range of problems [1,17–19]. Over the years, a number of different techniques have been developed for solving such systems.

2.1. Problem formulation

The general formulation for a 0–1 combinatorial optimisation problem is

$$\text{Minimise/Maximise} \quad \sum_{j=1}^{N} C_j X_j$$

$$\text{subject to} \quad A\,x \begin{cases} \leq \\ \geq \\ = \end{cases} B,$$

$$X_j \quad \{0,1\},$$

where

C is the cost vector of size $1 \cdot N$,

X is the solution vector of size $N \cdot 1$,

A is the constraint matrix of size $M \cdot N$,

B is the right-hand side vector of size $M \cdot 1$.

In this formulation, the cost must be minimised subject to the constraints specified in the matrix expression. The exact nature and values in the A and B matrices depend on the problem being modelled. For some problems, the A matrix contains constants arranged in a well-defined structure. Since the size of the X vector is usually quite large, there are an exponential number of potential solution states, although many of these may not satisfy the constraints. Accordingly, it is usually impractical to simply explore all 2^N possible states. Exact solution strategies and heuristic methods can be used to solve these problems. One of the most common generic exact solution tech-

niques is branch and bound [9,19]. Branch and bound explores the potentially huge set of values for the X vector by walking through nodes of a search tree. A branching heuristic is used to determine the order of assigning values to variables. By relaxing the integrality constraints on the problem, it is possible to achieve a lower bound on the cost, which is subsequently used to prune the search tree. If branch and bound manages to prune the search space effectively, then the optimal result will be returned. However, if it is unable to remove sufficient nodes, then it is only able to return an intermediate solution, which may be sub-optimal. In this case, the quality of the solution depends on the power of the branching heuristic in choosing a good order for processing the variables.

The heuristic (non-exact) approach takes either of two forms: a general purpose meta-heuristic, such as simulated annealing, genetic algorithms and tabu search, or a problem-specific heuristic. While the latter of the two classes can sometimes solve a specific problem quickly, it is difficult to adapt these approaches to different problems because they usually make use of very problem-specific information. The general meta-heuristic approach has the advantage that it can be applied to a wide range of problems with little modification; however, it is often slower than tailored methods. In this paper, we explore the use of simulated annealing for solving arbitrary integer problems.

2.2. Solving using simulated annealing

Simulated Annealing (SA) is a general purpose meta-heuristic method that has been successfully applied to a number of combinatorial optimisation problems [1,6,8, 14,20]. The theory of simulated annealing is derived from the physics of annealing substances. Simulated annealing seeks to minimise an energy function, which in combinatorial optimisation is the objective function. At the beginning of the annealing run, there is a high likelihood of accepting any transition made in the search space (whether it improves the solution or not) rather than later in the run. Each transition consists of choosing a variable at random and giving it another value (also at random). This process is done in accordance with an exponential acceptance function based on a parameter called temperature. The temperature is decremented until it is quite small and hence very few uphill moves (where a worse solution may replace the current solution) are accepted. As simulated annealing can make these uphill moves, settling into a local optimum is potentially avoided. The way the temperature is controlled is referred to as the cooling schedule.

Many of the applications of SA have been tailored to individual problems. GPSIMAN, on the other hand, was built for solving arbitrary 0–1 problems [7]. The algorithm accepts a conventional 0–1 linear formulation, as discussed in section 2.1. It moves through the search space by flipping variables at random and measuring the change in the cost function. This particular code implements a reheating schedule in

which a number of annealing runs are performed. The first of these runs calculates an appropriate initial and final temperature for the next run based on the range of objective costs it receives. This is a general approach to calculating temperature and is similar to that presented in other studies [1,17].

Figure 1 shows the overall algorithm used in GPSIMAN. The algorithm operates as follows. First, the problem model and SA parameters are initialised, and an initial solution is generated along with its cost. A number of annealing runs are subsequently performed. Within each run, variables are chosen and altered at random. A change is characterised by changing the state of a variable (flipping) and measuring the effect in terms of how the cost changes. Also, if the change causes the constraints to become infeasible, then feasibility is restored before the change in cost is evaluated.

The feasibility restoration technique flips variables (other than the original variable) in order to obtain a new feasible solution. The scheme employed in [7] is a heuristic technique whereby a score is computed for each of the variables based on how helpful a change in the variable value would be for feasibility restoration. The most helpful variable (the one with the highest score) is flipped and the resulting feasibility/infeasibility is recalculated. If feasibility has been restored, the procedure is terminated. However, in many instances, particularly for 0–1 problems which have many related constraints, this is not the case. The algorithm proceeds to calculate the next most helpful variable. This progresses as a depthwise tree search, in which the algorithm can backtrack should it find that the current sequence of flips cannot restore feasibility. This procedure is only useful if feasibility is relatively easy to restore; otherwise, the search for feasibility can quickly degenerate. If the process cannot restore feasibility after a fixed number of searches, then the original move is rejected. After feasibility is restored, the change in the objective cost can be calculated by simply adding or subtracting the appropriate coefficients (C_i) for all variables that have been flipped. If the cost is positive (for a minimisation problem), then the proposed solution is worse than the current one, and it is accepted as the current solution depending on the evaluation of Boltzmann's equation. If it is not accepted and a number of consecutive previous solutions have also been rejected, annealing is performed at the temperature at which the best solution (in this trial) was found. This continues for the remainder of this particular annealing run.

When the annealing run has ended, the temperature is reheated to a level slightly lower than that of the previous run. This entire process continues for the number of annealing trials specified by the user.

GPSIMAN has been tested on a number of 0–1 problems and achieved acceptable performance in comparison with a commercial branch and bound solver called SCICONIC. However, the code was only tested on a small range of specific 0–1 problems. In this paper, we have expanded the problem set using a number of well-established benchmark problems and have chosen some larger data sets. Also, we have tested the code against a more recent and more readily available IP solver, IBM's OSL.

Get SA parameters (T_I, T_F, *steps_per_trial*, *max_fails*)
Read problem model
Generate intial feasible solution (X)
$X_{orig} = X$
Compute cost of initial solution ($C(X)$)
DO user specified number of annealing trials BEGIN
    $\beta = (T_I - T_F)/(steps\_per\_trial \quad T_I \quad T_F)$
    $T = T_I$
    DO WHILE ($T > T_F$)
        $i = unif\_rand(1, size\ of\ (X))$
        $X' = X$
        $X'_i = 1 - X_i$
        Restore Feasibility (X')
        $\Delta C = C(X') - C(X)$
        $p = unif\_rand(0, 1)$
        IF ($\Delta C > 0$) **and** ($p > e^{-\Delta C/T}$) **THEN**
            $fails = fails + 1$
            IF ($fails > max\_ fails$) **THEN**
                $T_{min} = T$
                $\beta = 0$
                $T = T_{best}$
            ENDIF
        ELSE
            $X = X'$
            $fails = 0$
        ENDIF
        IF ($C(X) < C_{best}$) **THEN**
            $C_{best} = C(X)$
            $T_{best} = T$
        ENDIF
        $T = T/(1 + \beta T)$
    ENDDO
    $T_I = (T_I + T_{best})/2$
    IF ($\beta = 0$) **THEN**
        $T_F = (T_F + T_{min})/2$
    ELSE
        $T_F = T_F/2$
    ENDIF
    $X = X_{orig}$
ENDDO
END

Where:

unif_rand(*a*, *b*) returns a random uniformly distributed number between *a* and *b*.

Figure 1. Connolly's simulated annealing based 0–1 solver.

2.3. Test problem formulations

In this section, we describe some specific problems using a 0–1 ILP formulation technique. We then test GPSIMAN using these problems, and compare its performance to a commercial branch and bound code, OSL [9].

2.3.1. Quadratic Assignment Problem

The Quadratic Assignment Problem (QAP) is a facilities assignment problem. Each facility must be assigned to a unique location in order to minimise intercommunication cost between facilities. This problem is a common one in electronics, scheduling, manufacturing and parallel and distributed computing [16,18]. The 0–1 formulation of the QAP is

$$\text{Minimise} \quad \sum_{i=1}^{N}\sum_{a=1}^{N}\sum_{j=1}^{N}\sum_{b=1}^{N} C_{iaja}y_{iajb}$$

$$\text{subject to} \quad \sum_{i=1}^{N} x_{ia} = 1 \qquad 1 \le a \le N,$$

$$\sum_{a=1}^{N} x_{ia} = 1 \qquad 1 \le i \le N,$$

$$x_{ia} + x_{jb} \le y_{iajb} + 1 \qquad i < j, \quad a \ne b, C_{iajb} \ne 0,$$

$$x_{ij} \quad \{0, 1\} \qquad i, \quad j \; 1 \le i, j \le N.$$

In this formulation, x_{ij} is set to 1 if facility i is mapped to location j. The variable y_{ijkl} is set to 1 if facility i is mapped to location j and facility k is mapped to location l. The constraint for y enforces this logical conjunction. The problems from Nugent et al. [16] will be used here. Facility/location sizes of 5, 6, 7, 8, 10, 12, 15, 20 and 30 are considered.

2.3.2. Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is where a salesman must visit each of a number of cities exactly once. The salesman starts and ends at a base city and the solution is therefore called a tour. The objective is to minimise the length of the tour that the salesman makes. There are a number of applications of the TSP, some of which include circuit board design and transport routing [15]. In this paper, we use the following 0–1 formulation of the TSP:

$$\text{Minimise} \quad \sum_{i=2}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N} d_{jk}y_{ij(i-1)k} + \sum_{j=1}^{N}\sum_{k=1}^{N} d_{jk}y_{1jNk}$$

$$\text{subject to} \quad \sum_{k=1}^{N} x_{ik} = 1 \qquad 1 \le i \le N,$$

$$\sum_{i=1}^{N} x_{ij} = 1 \qquad 1 \le j \le N,$$

$$x_{ik} + x_{jl} \le y_{ijkl} + 1 \qquad \text{IF } i = N \text{ THEN } j = 1 \text{ ELSE } j = i + 1,$$

$$k \ne l \; 1 \le i, j, k, l \le N,$$

$$x_{ij} \quad \{0,1\} \qquad i, \quad j \; 1 \le i, j \le N.$$

This formulation differs from the more conventional one which includes a number of sub-tour constraints [15]. This representation was chosen because it has fewer constraints than the more usual sub-tour form. It uses a variable x_{ij} which is set to 1 if city j is the ith city visited in the tour. The variable y_{ijkl} is set to 1 if city j is the ith visited and city l is the kth visited. The constraint for y enforces the logic operation of conjunction between neighbouring cities. City sizes of 14, 29, 48, 96, 280 and 666 are considered.

2.3.3. Graph colouring

Given a graph G, where $G = (V, E)$ (i.e. the graph consists of a number of vertices connected together by edges), the aim is to assign a colour to each vertex, such that the colour of the vertex is different from that of its neighbours (those vertices to which it is connected by an edge). The aim is to map a minimum set of colours onto the graph. The 0–1 formulation of the graph colouring problem is

$$\text{Minimise} \quad \sum_{k=1}^{K} c_k$$

$$\text{subject to} \quad \sum_{k=1}^{K} x_{ik} = 1 \qquad 1 \le i \le N,$$

$$x_{ik} + x_{jk} \le 1 \qquad k, \quad i, j \text{ s.t. an edge } i \to j \text{ in } G,$$

$$\sum_{i=1}^{N} x_{ik} \le N \quad c_k \qquad 1 \le k \le K,$$

$$c_k, x_{ik} \quad \{0,1\} \qquad i, \quad k \; 1 \le i \le N, 1 \le k \le K.$$

2.3.4. Bin packing

A set of items, each of which has a particular weight, must be fitted into a number of bins. Each bin has the same weight capacity. The aim is to use as few bins as possible. The 0–1 formulation of the bin packing problem is

$$\text{Minimise} \quad \sum_{j=1}^{M} c_j$$

$$\text{subject to} \quad \sum_{j=1}^{M} x_{ij} = 1 \qquad 1 \le i \le N,$$

$$\sum_{i=1}^{N} w_i x_{ij} < W_{max} \qquad 1 \le j \le M,$$

$$\sum_{i=1}^{N} x_{ij} \le N \quad c_j \qquad 1 \le j \le M,$$

$$c_j, x_{ij} \quad \{0,1\} \qquad i, \quad j \; 1 \le i \le N, \; 1 \le j \le M.$$

In this formulation, the variable x_{ij} is set to 1 if item i is in bin j. These problems were obtained through the World OR library [5]. The size of the problem depends on how many items there are. Sizes of 120, 250 and 500 items are used.

2.3.5. Generalised Assignment

The Generalised Assignment Problem (GAP) is a problem in which n jobs must be assigned to m agents, subject to each job being assigned to only one agent and the capacity constraints of the agents are not violated. The objective is to minimise the total cost of the assignment of jobs to agents. The formulation of GAP is

$$\text{Minimise} \quad \sum_{i=1}^{N} \sum_{j=1}^{M} c_{ij} x_{ij}$$

$$\text{subject to} \quad \sum_{i=1}^{N} a_{ij} x_{ij} < b_j \qquad 1 \le j \le M,$$

$$\sum_{j=1}^{M} x_{ij} = 1 \qquad 1 \le i \le N,$$

$$x_{ij} \quad \{0,1\} \qquad i, \quad j \; 1 \le i \le N, \; 1 \le j \le M.$$

In this formulation, the variable x_{ij} is set to 1 if job i is assigned to agent j. These problems were obtained through the World OR library [5]. The size of the problem varies depending on the number of jobs and agents, and varies from 15 jobs to 32 jobs with 5 to 8 agents.

3. Integer simulated annealing

3.1. Integer formulation

The 0–1 formulation technique discussed in section 2 has many advantages. The most significant of these is that it is possible to model many different problems using

a set of linear equations. Consequently, the continuous relaxation of the system can be solved with a linear algorithm like the simplex method, and this may be used to help find the solution to the discrete problem. The main disadvantage is that a large number of variables and constraints are necessary to represent some very simple situations. For example, in order to represent N different mappings, N variables are required instead of $\log_2(N)$. Further, a constraint is required to guarantee that only one variable is set at a time. In the context of GPSIMAN, this means that the feasibility restoration logic must be invoked each time a fairly simple move is made. On the other hand, an integer formulation only requires one variable for encoding the N states using $\log_2(N)$ bits. More importantly, no constraints are necessary for enforcing the correct encoding. Profiling GPSIMAN indicated that a substantial amount of time was spent restoring these encoding constraints. Given these shortcomings, we have investigated a formulation technique which allows variables to take arbitrary integer values, and have explored representations of the problems discussed in section 2.3. The rationale behind this is to try and minimise the amount of feasibility restoration which is performed. The problems examined in this paper appear to reduce to three main classes of constraints, namely, order based, arbitrary linear constraints and assignment problems. The main observations are that

- fewer variables are required in the integer formulation,
- all of the equality and encoding constraints are removed,
- it is necessary to allow inequality ($\neq$) constraints, which are nonlinear,
- more complex cost functions are required, often involving nonlinear operators like *maximum*.

At first examination, the introduction of these non-linearities would appear to outweigh the advantages of fewer variables and constraints. However, simulated annealing can be applied to nonlinear problems as well as linear, and the experimental evidence presented in section 4 shows that the new scheme performs well. Bearing this in mind, there are many tailored heuristics and finely tuned SA implementations that will solve particular combinatorial optimisation problems very efficiently (more so than INTSA). An example of this is the Kernighan–Lin algorithm for the TSP. However, to adapt these approaches to different problems often requires complex redesign of the algorithm and hence may be impracticable. Therefore, the importance of INTSA and other general solvers cannot be underestimated.

From the study so far, many 0–1 formulated problems are amenable to the above categorisation scheme and subsequent reformulation. Problems which are not easy to reformulate using this categorisation include those which are naturally constructed using binary variables, like the well-known knapsack and set partitioning problems. Care must also be taken when reformulating problem models, as a number of different but equivalent representations may be possible. The designer must think carefully about the objective function and the type of neighbourhood necessary for the problem.

Iterative techniques such as SA work best in an undulating cost landscape, as the variation in cost guides the search towards better solutions. For instance, an alternative representation for the graph colouring problem might be to minimise the number of edges in which both nodes have the same colour. This will lead to a cost landscape that is "hillier" than that of the models presented in sections 2.3.3 and 3.3.3. However, this will make little difference for the relatively sparse graphs considered in this study.

3.1.1. Order based constraints

These are problems such as the TSP and QAP in which the cost is evaluated in terms of the *order* of the solution vector. Generally, each element of the solution vector x must have a distinct value from every other element. These problems have the general form

$$\begin{aligned} &\text{Minimise} && f(x) \\ &\text{subject to} && x_i \neq x_j \qquad i,\ j\ (i \neq j), \end{aligned}$$

which can be written in matrix form as

$$Ax \neq b,$$

where A is a rectangular matrix which has entries of 1, 0 and –1 covering all possible combinations of pairs of the x vector.

Given this problem structure, the most appropriate move operator from one point in the feasible region to another is n-opt. We have investigated 2-opt (a simple swap) and 3-opt, both of which ensure that the constraints are not violated.

3.1.2. Arbitrary linear constraints

These are problems in which the A matrix can take on an arbitrary linear form (with the augmentation of the ≠ operator). An example of this is the graph colouring problem. The problems are of the form

$$\begin{aligned} &\text{Minimise} && f(x) \\ &\text{subject to} && Ax \begin{cases} \leq \\ \geq \\ \neq \\ = \end{cases} b. \end{aligned}$$

The A matrix can be of arbitrary form. The move operator should ensure that the solution it produces is feasible, as feasibility cannot be so easily guaranteed as it can with the order type problems.

3.1.3. Assignment problems

These are problems in which a specified list of items must be allocated to a number of entities. An example of this is the bin packing problem, in which items are

assigned to bins such that no item is assigned to be in more than one bin. The general form is

$$\text{Minimise} \quad f(x)$$

$$\text{subject to} \quad w \; \Delta \begin{cases} \leq \\ \geq \\ \neq \\ = \end{cases} b,$$

where

w is a constant vector,

$\Delta_{ij} = \delta_{x_i j}$,

$$\delta_{x_i j} = \begin{cases} 1 & \text{if } x_i = j, \\ 0 & \text{if } x_i \neq j. \end{cases}$$

In this formulation, Δ is a function of x, and therefore, as x changes, so will the Δ matrix. Δ effectively decodes the encoded integer values in x into binary variables and places these values in each row of Δ. However, unlike the 0–1 formulation, no constraints are needed to enforce the encoding, which is maintained by the integer move operator. The move operator does need to ensure that the assignment of items to entities does not violate the other constraints, and this is discussed in the next section.

3.2. Solving using simulated annealing

The integer version of simulated annealing has the same structural properties as the algorithm presented in figure 1. Apart from operating in integer (rather than 0–1) space, it differs in the way feasibility is maintained, which is handled differently depending on the categorisation. As discussed in section 3.1.1, once initial feasibility has been established in the order-based problems, it can be maintained by an n-opt operator. Accordingly, variables are interchanged and thus feasibility is maintained. In the other two formulations, a change to an individual variable has the potential to cause infeasibility, unless some other variables are also altered. The scheme which has been implemented first attempts to make a move without introducing any infeasibility, and then if this is not possible, it restores feasibility.

There are two obvious ways of implementing a feasibility maintenance scheme:

(a) Calculate a set of feasible values that the variable can take without violating any of the constraints. This involves evaluating each of the constraints and eliminating values from the possible set iteratively. An element can be chosen at random from the final set.

(b) Choosing a value for the variable and then testing whether it violates any of the constraints. This process is repeated until all possible values have been tested or a suitable value is chosen.

We have found that the latter version requires less computational effort; however, the former approach may be useful for highly constrained problems where few values satisfy the constraints. If the problem is too highly constrained, then the scheme fails to generate any moves. Feasibility maintenance may therefore cause the search to converge slowly to the optimum, especially if it is obscured by infeasible space. In order to traverse this space, the algorithm subsequently chooses another variable to alter in an attempt to restore feasibility, using the same system described in (b). This process can be applied recursively in an attempt to restore feasibility; however, we have found that all of the problem could be restored using one level of recursion or less.

3.3. Test problem formulations

Reformulating the 0–1 problems into the integer form requires some skill. In this section, we present the results of the reformulation. We also introduce some new nonlinear cost functions where required, including $uniq(x)$ which returns the number of unique values in the vector x.

3.3.1. Quadratic Assignment Problem

In this problem, we replace the binary variables with a new integer variable, x, which denotes the mapping between each facility and location. Specifically, if $x_i = k$, then facility k is mapped to location i. The variable f_{ij} represents the amount of traffic between two facilities, and d_{ij} is the distance between them.

$$\begin{aligned} \text{Minimise} \quad & \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} f_{x_i x_j} d_{ij} \\ \text{subject to} \quad & x_i \neq x_j \quad 1 \leq i,\ j \leq N\ (i \neq j), \\ & 1 \leq x_i \leq N, 1 \leq i \leq N. \end{aligned}$$

3.3.2. Travelling Salesman Problem

In this problem, the integer variable x is used to denote the order in which the cities of the tour are visited. Specifically, if $x_i = k$, then city k is the ith city visited on the tour. The variable d_{ij} represents the distance between two cities, i and j.

$$\begin{aligned} \text{Minimise} \quad & \sum_{i=2}^{N} d_{x_i x_{i-1}} \\ \text{subject to} \quad & x_i \neq x_j \quad 1 \leq i,\ j \leq N\ (i \neq j), \\ & 1 \leq x_i \leq N, 1 \leq i \leq N. \end{aligned}$$

3.3.3. Graph colouring

In this problem, the integer variable x is used to hold the colour of each node in the graph. Specifically, if $x_i = k$, then node i has colour k. The constraints guarantee that no adjacent nodes have the same colour, and the cost function minimises the number of colours allocated to the graph. Since more than one node can have the same colour if they are not adjacent, the *uniq* function is used to measure the number of colours allocated to the entire graph.

$$\begin{aligned} &\text{Minimise} && uniq(x) \\ &\text{subject to} && x_i - x_j \neq 0 \qquad i, j \text{ such that an edge } i \rightarrow j \text{ in } G, \\ & && 1 \leq x_i \leq K,\ 1 \leq i \leq N. \end{aligned}$$

3.3.4. Bin packing

In this problem, the integer variable x is used to denote the packing of items to bins. Specifically, if $x_i = k$, then item i is packed in bin k. As in the graph colouring problem, *uniq* is used to measure the number of bins used in the allocation. The value of d_{ij} is set to 1 if $i = j$, otherwise it is 0. Accordingly, item i is mapped to bin j, in which case the weight for the item is added to the relevant constraint.

$$\begin{aligned} &\text{Minimise} && uniq(x) \\ &\text{subject to} && \sum_{i=1}^{N} \delta_{x_i j} \ \ w_i < W_{max} \qquad 1 \leq j \leq M, \\ & && 1 \leq x_i \leq M,\ 1 \leq i \leq N. \end{aligned}$$

3.3.5. Generalised assignment

This problem is similar to bin packing, except that the aim is to minimise the cost of mapping jobs to agents, rather than minimising the number agents per se. In this case, the integer variable x contains the agents for each job. Specifically, if $x_i = k$, then job i is assigned to agent k. The cost variable c_{ij} measures the cost of assigning job i to agent k.

$$\begin{aligned} &\text{Minimise} && \sum_{i=1}^{N} c_{ix_i} \\ &\text{subject to} && \sum_{i=1}^{N} \delta_{x_i j} \ \ a_{ij} < b_j \qquad 1 \leq j \leq M, \\ & && 1 \leq x_i \leq M,\ 1 \leq i \leq N. \end{aligned}$$

4. Results

In this section, we present the results of running GPSIMAN, INTSA and OSL on a range of problems, namely bin packing, graph colouring, quadratic assignment,

Table 1

Problem instances used in this study.

Problem name	Problem type	Problem size	Optimal cost
bin1	Bin packing	120 items	50 bins
bin1a	Bin packing	120 items	50 bins
bin2	Bin packing	250 items	99 bins
bin2a	Bin packing	250 items	99 bins
bin3	Bin packing	500 items	201 bins
bin3a	Bin packing	500 items	198 bins
graph1	Graph colouring	50 nodes, 150 edges	4 colours
graph1a	Graph colouring	50 nodes, 150 edges	4 colours
graph2	Graph colouring	100 nodes, 300 edges	4 colours
graph2a	Graph colouring	100 nodes, 300 edges	4 colours
graph3	Graph colouring	250 nodes, 550 edges	4 colours
graph4	Graph colouring	300 nodes, 740 edges	4 colours
qap1	QAP	5 facilities/locations	25
qap2	QAP	6 facilities/locations	43
qap3	QAP	7 facilities/locations	74
qap4	QAP	8 facilities/locations	107
qap5	QAP	12 facilities/locations	289
qap6	QAP	15 facilities/locations	575
qap7	QAP	20 facilities/locations	1285
qap8	QAP	30 facilities/locations	3062
tsp1	TSP	14 cities	26 miles
tsp2	TSP	29 cities	9063 miles
tsp3	TSP	48 cities	33503 miles
tsp4	TSP	97 cities	469 miles
tsp5	TSP	280 cities	2557 miles
tsp6	TSP	666 cities	3693 miles
gap1	GAP	5 agents, 15 jobs	261
gap2	GAP	5 agents, 20 jobs	277
gap3	GAP	5 agents, 25 jobs	438
gap4	GAP	5 agents, 30 jobs	423
gap5	GAP	8 agents, 24 jobs	403
gap6	GAP	8 agents, 32 jobs	525

travelling salesman and generalised assignment. A description of each problem (as well as a symbolic label) is given in table 1. The cooling schedule for each problem was calculated according to the general scheme presented in section 2.2. However, trial and error was used to determine the most effective annealing run length for each problem instance. The OSL code was optimised in terms of computational performance for ILP problems (as specified in the OSL manual [9]). The problems were all run on an IBM RS6000 Model 590.

Table 2

RS6000 processing time to reach the optimal solution.

Problem name	GPSIMAN time (secs)			INTSA (secs)			OSL time (secs)	
	Min	Ave	Max	Min	Ave	Max	Opt	Exhaust
bin1	274	580	1283	< 1	94	165	2	125
bin1a	309	603	1881	5	48	108	10	69
bin2		CNS			CNS			CNS
bin2a		CNS			CNS			CNS
bin3		CNM			CNS			CNS
bin3a		CNM			CNS			CNS
graph1	3	13	45	< 1	< 1	< 1	32	901
graph1a	3	3	4	< 1	3	10	29	939
graph2	10	122	351	< 1	7	15	1632	CNS
graph2a	50	227	547	< 1	13	35	1946	CNS
graph3	409	1308	2770	6	155	432	1515	CNS
graph4	350	2502	5639	211	2916	6508	2958	CNS
qap1	19	343	774	< 1	< 1	< 1	2	12
qap2	41	964	3704	< 1	< 1	< 1	9	89
qap3		CNS		< 1	< 1	< 1	725	1737
qap4		CNS		< 1	< 1	< 1	4708	5544
qap5		CNS		< 1	< 1	2		CNS
qap6		CNS		< 1	1	4		CNS
qap7		BIG		7	503	1106		CNS
qap8		BIG		720	2555	6600		CNS
tsp1		CNS		< 1	< 1	< 1		CNS
tsp2		CNM		2	6	12		CNS
tsp3		CNM			CNS			CNS
tsp4		BIG			CNS			BIG
tsp5		BIG			CNS			BIG
tsp6		BIG		5411	5423	5427		BIG
gap1	6	15	48	< 1	< 1	< 1	3	4
gap2	884	3283	4880	< 1	< 1	< 1	3	3
gap3		CNS		< 1	4	9	21	23
gap4		CNS		3	16	31	25	44
gap5		CNS		2	5	9	< 1	10
gap6		CNS		3	13	21	13	128

Table 2 presents the CPU time required (in seconds) for each of the algorithms. For the simulated annealing runs (GPSIMAN and INTSA), the minimum, average and maximum of 10 independent runs is reported. For the OSL runs, the time to find the optimal solution and the time to exhaustively search (with pruning) the space is reported. CNS indicates that the algorithm Could Not Solve the problem to optimality within 2 hours of CPU time. CNM indicates that GPSIMAN Could Not Make a move

Table 3

Problems that GPSIMAN could not solve to optimality.

Problem name	Worst GAP (%)	GPSIMAN time (secs)			Integer SA time (secs)		
		Min	Ave	Max	Min	Ave	Max
bin2	21.2	5407	5574	5833	< 1	< 1	< 1
bin2a	21.2	5173	5200	5223	< 1	< 1	< 1
qap3	20	2	2	2	< 1	< 1	< 1
qap4	21	68	210	331	< 1	< 1	< 1
qap5	20	5004	5801	CNS	< 1	< 1	< 1
qap6	29.4	3	3	3	< 1	< 1	< 1
tsp1	15	349	520	710	< 1	< 1	< 1
gap3	4.4	953	1090	1348	< 1	1	2
gap4	3.5	845	1025	1143	3	3	3
gap5	0.7	129	190	297	20	107	205
gap6	3.6	1362	1628	1987	2	3	3

Table 4

Problems that INTSA could not solve to optimality.

Problem name	Best GAP (%)	Integer SA time (best) (secs)
bin3	3.5	16
bin3a	4.3	22
tsp3	2.5	18
tsp4	5.5	25
tsp5	8.2	1009

using its feasibility restoration technique. BIG indicates that the problem exceeded the memory and disk space storage capacity of the hardware/software platform.

The results indicate that INTSA outperforms both GPSIMAN and OSL for almost all problems in which all three algorithms can find the optimal solution. Further, INTSA finds some optimal solutions which cannot be found by GPSIMAN and OSL.

Some problems could not be solved to optimal by any of the algorithms. In table 3, we report the results for problems that GPSIMAN could not solve to optimality but which were solved by INTSA. In these cases, we report the gap between the worst GPSIMAN problem and optimal, and show the times taken by both annealing codes to achieve the same quality. In almost all of these cases INTSA achieves the same quality solution as GPSIMAN in less than one CPU second. Finally, in table 4, we present results for the problems which could not be solved by either annealing code or

OSL. In these cases, we report the best result achieved by INTSA and the gap between this solution and optimal. The results indicate that it is possible to achieve reasonably tight upper bounds for most of the problems.

5. Conclusions

In this paper, we have presented a scheme of simulated annealing which operates on integer variables (INTSA). We have compared this against GPSIMAN which accepts arbitrary 0–1 ILP formulations. INTSA outperforms GPSIMAN on almost all of the problems that we tested, and also outperforms the branch and bound code from OSL. This can be attributed to a number of factors. First, the equivalent integer formulations have fewer variables than their 0–1 counterparts. Second, many constraints which enforce the encoding of the 0–1 variables are removed. Third, it is possible to design annealing moves which minimise the amount of feasibility restoration that must be performed. Such move operators take account of the constraint structure and also restrict the target variable values to ones which yield feasible solutions.

In principle, a general solver based on this integer formulation technique could accept an arbitrary problem model and proceed to solve the problem without the need for the user to supply special purpose software modules. This is achieved by using an augmented algebraic form coupled with transition operators that have only been available for special purpose SA and heuristic codes. Therefore, this method is different from other tailored implementations of SA. In these, the structure of the problem is analysed by the practitioner in order to determine the most appropriate internal representation of the problem and neighbourhood transition operators. The implementation is then expressed in code. Whilst both the tailored and the toolbox techniques may produce more efficient code, a considerable amount of time is required to develop an implementation that solves a particular problem.

The specification developed differs from a 0–1 based linear formulation technique in a few respects. First, it allows nonlinear operators in the cost function. Second, it utilises the $\neq$ operator in constraints. Third, it uses an unusual technique for specifying assignment type of problems based on a transformation of the solution vector. The three classifications seem to be capable of representing a wide range of problems; however, it remains to be shown that it is possible to represent all problems that can be written using a 0–1 technique.

The major disadvantage of INTSA is that it uses an unconventional specification technique. We are exploring ways of automatically converting 0–1 problems into the integer form. This may be possible by examining the constraints and detecting groupings of variables which could be encoded into simple integers. However, at this stage, this remains an open ended problem.

The results reported in this paper were actually gathered from a number of different programs, each of which handled one of the classes of constraints, rather than one program which could differentiate the constraint class and choose the appropriate algorithms. We are planning to combine these into one program in the near future.

We are also investigating the design of new computer architectures which are tailored to executing algorithms like INTSA. It is possible to gain a performance improvement in the order of 100 times by using such special purpose architectures [2–3]. Such systems would yield a very powerful tool for practitioners because they would allow model developers to experiment very quickly with different problem formulations and data sets.

References

[1] D. Abramson, H. Dang and M. Krishnamoorthy, A comparison of two methods for solving 0–1 integer programs using a general purpose simulated annealing, Annals of Operations Research 63(1996)129–150.

[2] D. Abramson, A. de Silva, M. Randall and A. Postula, Special purpose computer architectures for high speed optimisation, in: *Proceedings of the 2nd Australasian Conference on Parallel and Real Time Systems*, 1995, pp 13–20.

[3] D. Abramson, A very high speed architecture to support simulated annealing, IEEE Computer (May, 1992)27–34.

[4] D. Abramson, P. Logothetis, M. Randall and A. Postula, Application specific computers for combinatorial optimisation, in: *The Australian Computer Architecture Workshop*, Sydney, Feb. 1997.

[5] J. Beasley, OR-library: Distributing test problems by electronic mail, Journal of the Operational Research Society 41(1990)1069–1072.

[6] N. Collins, R. Eglese and B. Golden, Simulated annealing: An annotated bibliography, American Journal of Mathematical and Management Sciences 8(1988)209–307.

[7] D. Connolly, General purpose simulated annealing, Journal of the Operational Research Society 43 (1992)495–505.

[8] R. Eglesse, Simulated Annealing: A tool for operational research, European Journal of Operational Research 46(1990)271–281.

[9] IBM, Optimisation Subroutine Library User Manual, IBM Corporation, 1990.

[10] L. Ingber, Simulated annealing: Practice versus theory, Computer Modelling 18(1993)29–57.

[11] D. Johnson, C. Aragon, L. McGeogh and C. Scheveon, Optimisation by simulated annealing: An experimental evaluation, Part I: Graph partitioning, Operations Research 37 (1991) 865–892.

[12] D. Johnson, C. Aragon, L. McGeogh and C. Scheveon, Optimisation by simulated annealing: An experimental evaluation, Part II: Graph colouring and number partitioning, Operations Research 39(1991)378–406.

[13] S. Kirkpatrick, D. Gelatt and M. Vecchi, Optimization by simulated annealing. Science 220(1983) 671–680.

[14] C. Koulamas, S. Antony and R. Jansen, A survey of simulated annealing to operations research problems, Omega International Journal of Management Science 22(1994)41–56.

[15] E. Lawler, *The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimisation*, Wiley, 1990.

[16] C. Nugent, T. Vollman and J. Runl, An experimental comparison of techniques for the assignment of facilities to locations, Operations Research 16(1968)150–173.

[17] I. Osman, Heuristics for the generalised assignment problem: Simulated annealing and tabu search approaches, OR Spektrum 17(1995)211–225.

[18] P. Pardalos and H. Wolkowicz, Quadratic assignment problems and related problems, in: *Discrete Mathematics and Theoretical Computer Science*, Dimacs, 1994.

[19] H. Taha, *Operations Research: An Introduction*, 5th ed., Macmillan, 1992.

[20] P. van Laarhoven and E. Aarts, *Simulated Annealing: Theory and Applications*, D. Reidel, 1987.