Algorithms and Heuristics for Combinatorial Optimization Problems: Case Study of Academic Course Scheduling

by

M. A. Saleh Elmohamed

Abstract of Dissertation

July, 1999

Abstract

Algorithms and Heuristics for Combinatorial Optimization Problems: Case Study of Academic Course Scheduling

by

M. A. Saleh Elmohamed
M.S., Syracuse University, 1994

Dissertation

July, 1999

ALGORITHMS AND HEURISTICS FOR COMBINATORIAL OPTIMIZATION PROBLEMS: CASE STUDY OF ACADEMIC COURSE SCHEDULING

By

M. A. Saleh Elmohamed

M.S., Syracuse University, 1994

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in computer science and engineering in the Graduate School of Syracuse University

July, 1999

Approved ____________________________

Dr. Geoffrey Fox

Date ____________________________

Abstract

Acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Summary

In this dissertation we try to answer the following question: How do we go about solving optimization problems, in general, and academic course scheduling (or timetabling) at the university level, in particular? we give a detailed answer to the latter part, and shed some light on the case of tackling optimization problems in general.

To summarize, we have investigated a variety of optimization algorithms, in particular, random search, hill climbing, knowledge-based systems (or expert systems), tabu search and simulated annealing-based methods to solve the course scheduling problem. The simulated annealing-based approach includes mean-field annealing, simulated annealing with three different cooling schedules, and the use of a rule-based preprocessor to provide a good initial solution for annealing.

The best results were obtained using simulated annealing with adaptive cooling and reheating as a function of cost, and a rule-based preprocessor. This approach enabled us to obtain valid schedules for the timetabling problem for a large university, using a complex cost function that includes student preferences. None of the other methods were able to provide a complete valid schedule.

1.2 Objectives

The primary objective of this dissertation is to shed light on how we go about solving optimization problems in general, and to present detailed analysis on deriving an approximate solution to the problem of university course scheduling in particular.

In order to solve an optimization problem such as the TSP, we design a series of steps, or a procedure, which we can problem instance to which it claims to adhere and, ideally, be sure of getting a good answer in a given amount of time*. Such a procedure is known as an 'optimization algorithm'.

Sometimes we may be happy with (or forced to accept) a less than optimal solution, in which case it may be easier to find a suitable algorithm. We would like our algorithms to return as good an answer as possible in as little time as possible, and these will be important comparative judges of the merit of different algorithms. Again, this is more subtle than it seems in that the time will vary from problem instance to another, and we have to decide whether we want to minimize the best case, the worst case or the average case. The reason NP-complete problems are considered difficult is that all known algorithms which guarantee the optimal solution will operate in exponential time in the size of the problem in the worst case scenario. For reasonably large problem instances this is impractical and so we reduce our expectations of the solution from 'optimal' to 'good', and seek algorithms which can still search effectively in these difficult search spaces. As well as average versus best/worst case in time-requirements, we also want to understand average versus best/worst case in quality of solution found.

The optimization problem we have closely tackled is university course scheduling (or timetabling), which can be summarized as follows:

Given data sets of classes and their days, enrollments, and instructors; rooms and their capacities, types, and locations; distances between buildings; priorities of each building for different departments; and students and their class preferences; the

*Usually we can avoid complications due to the Halting problem, as we address problems of a simpler nature in which we are really fighting against NP-completeness and are sure that a solution does exist.

problem is to construct a feasible class schedule satisfying all the hard constraints and minimizing the medium and soft constraints. Hard constraints are space and time constraints that must be satisfied, such as scheduling only one class at a time for any teacher, student, or classroom. Medium and soft constraints are student and teacher preferences that should be satisfied if possible.

1.3 Approximation Algorithms

When facing an *NP*-complete optimization, we may want to consider algorithms that do not produce optimum solutions, but solutions *guaranteed to ve close to the optimum.* Suppose that we wish to obtain such solutions for an optimization problem, maximization or minimization. For each instance x of this problem, there is an optimum solution with value $opt(x)$; let us assume that $opt(x)$ is always a positive integer.

Suppose now that we have a polynomial algorithm A which, when presented with instance x of the optimization problem, returns some solution with value $A(x)$. Since the problem is *NP*-complete and A is polynomial, we cannot realistically hope that $A(x)$ is always the optimum value. But suppose that we know that the following inequality hold:

$$\frac{|opt(x) - A(x)|}{opt(x)} \leq \epsilon \tag{1.1}$$

where ϵ is some positive real number, hopefully very small, that bounds from above the worst-case relative error of algorithm A. (The absolute value in this inequality allows us to treat both minimization and maximization problems within the same framework.) If algorithm A satisfies this inequality for all instances x of the problem, then it is called an **ϵ-approximation algorithm.**

Once an optimization problem has been shown to be *NP*-complete, the following question becomes most important: Are there ϵ-approximation algorithms for this problem? And if so, how small can ϵ be? Let us observe at the outset that such questions are meaningful only if we assume that $p \neq NP$, because, if $P = NP$, then the problem can be solved exactly, with $\epsilon = 0$.

1.4 Type of problems addressed

A question to ask here is that: What problems can stochastic optimization algorithms such as simulated annealing address?

Most of the algorithms we have looked at adhere to a simple set of requirements upon the space of problems to which they apply. In fact we require no more of a problem than that:

1. we can generate a random solution,
2. a 'local perturbation' to a solution is defined,
3. such perturbations over all relevant parts of the solution space (this is usually naturally satisfied).

All of the algorithms discussed below fall within this category. Also note that because we search directly in the problem space, all the algorithms we consider will return both the actual solution and its cost (or value of utility depending upon the context). One can also envisage algorithms which only return a bound upon the solution cost, and not the actual solution. Whilst of theoretical value, in most practical instances we would ordinarily require the solution as well. Finally, it is important to note that:

- individual problem instances may vary enormously in difficulty. The average versus best or worse case difficulty (or complexity) of search and problem solving may depend in quite particular fashions on the method of problem selection. These issues were addressed later in greater depth.
- these algorithms may not specify a natural termination condition, and in such cases heuristic principles must be used. Possibilities include: maximum running time, solution quality, time since last improvement; or combinations thereof.

In the descriptions which follow, we shall assume we are supplied with a function 'FITNESS: $\sum \rightarrow R$' which supplies us with the solution quality. Without loss of generality, we assume higher fitnesses are better.

1.5 Combinatorial Optimization

Combinatorial optimization refers to the problem of minimizing a function of a large but finite set of variables. The cost function or energy function $E(\vec{x})$ assigns a real number to each possible state $\vec{x}$ of the system. The configuration space X is given by the finite set of all possible system configurations $\{\vec{x}\}$. The problem is then that of finding the global minimum of $E(\vec{x})$ over all $\vec{x}$ in X.

The existence of a large number of local but not global minima of $E(\vec{x})$ in configuration space conspires against the success of heuristic optimization methods. Algorithms such as simulated annealing is an improvement upon downhill algorithms in that it provides a mechanism for getting out of such local minima. It should be noted that while the notion of global minima is an absolute one that depends only on the cost function $E(\vec{x})$, the concept of local minima is relative to the topology since it implies comparing the value of $E(\vec{x})$ with that of $E(\vec{x}^{'})$ for those $\vec{x}^{'}$ which are neighbors of $\vec{x}$.

The topology of configuration space is defined by the neighbor relation. A natural definition of neighborhood for optimiztion problems is given by the move set: a small rearrangment of the system that produces a new trial configuration from the present one. Distance $d(\vec{x}, \vec{x}^{'})$ between two states is then defined as the minimum number of moves needed to turn one into other; and the neighborhood of a given state $\vec{x}$ is the set of all states $\vec{x}^{'}$ which can be reached from $\vec{x}$ with one move, i.e. $d(\vec{x}, \vec{x}^{'}) = 1$.

1.6 Techniques for Constraint-Directed Scheduling

Christopher Beck, in his PhD thesis [?], discusses among other things, constraint-directed techniques that have been applied to scheduling. These are heuristic commitment technigues, propagators, and retraction techniques. In the discussion he uses the term *commitment* to refer to a decision made during search. It is referred to or conceptualized as a constraint that is added to or removed from the constraint graph. For example, assigning a start time to an activity can be modeled as the assertion of a unary equals commitment on the activity, while sequencing two activities can be represented as the assertion of a binary precedence constraint between the two activities.

1.6.1 Heuristic Commitment Techniques

A heuristic commitment technique is a procedure that finds new commitments to be added to the graph in order to (heuristically) move toward a solution. In traditional constraint-directed search (and, indeed, artificial intelligence search more generally) heuristic commitments are the primary component of a search for a solution. A critical component of constraint-directed scheduling techniques, therefore, is the technique used to generate and select the heuristic commitments to make. We will be looking at a number of heuristics that have been applied to constraint-directed scheduling.

1.6.2 Intuitions on Heuristic Performance

Here we present intuitions about the nature of heuristic commitment techniques. These intuitions are speculative and their investigation forms the basis of our future work on heuristics.

We believe that there are two characteristics that correlate highly with the success of a heuristic commitment technique.

1. The Short dead-end Proof Intuition: the ability of a heuristic to quickly discover that the search is at a dead-end. Imagine a search that progresses through a series of states to state S. Assume that there are no solutions in the subtree below S, but that we are unable to prove this at state S. Some heuristic commitments must be made to explore the subtree below S in order to prove that S is a dead-end. A heuristic that is able, through its heuristic commitments and subsequent propagation, to prove a dead-end while exploring a small number of states in the subtree will be superior to on that must explore a larger subtree.

2. The Few Mistakes Intuition: a high likelihood of making a commitment that leads to a solution. A heuristic that has a higher likelihood of making a commitment that leads to a solution, that is one that does not result in a dead-end, is likely to be a higher quality heuristic than one with a lower probability of making the correct decision.

These two characteristics are independent from the perspective that if either characteristic were infallible, search performance would be independent of the other characteristic. If a heuristic never made a commitment that resulted in a dead-end state, the ability to detect such a state is irrelevant. Similarly, if a search could immediately detect that it was a dead-end, then the quality of the commitments is irrelevant: any mistake is immediately detected and repaired.

Withe real, fallible heuristics, these two characteristics interact: it may be the case that a heuristic with a high probability of making the right decision results in a relatively large effort in dicovering when it has made a mistake. Conversely, a heuristic with a lower probability of making a right decision may be able to find a shorter proof of a dead-end (on average) and therefore recover from mistakes quickly. The best heuristic will be the one that minimizes the size of the overall search tree and this may well be achieved by some trade-off between the two characteristics.

1.7 Heuristic and Algorithms

How in general should we go about designing an algorithm for a particular purpose? Looking at the above algorithms, we see they are derived from a wide array of fields of knowledge:

- Random Search – a no-knowledge approach;
- Hill Climbing – the most basic idea given the concept of a fitness landscape;
- Genetic Algorithms – based upon evolution and the power of recombination of partial solutions;
- Tabu Search – an extension of classical local improvement heuristics. The search is conducted in a search space made up of all possible solutions considered;
- Simulated Annealing – based upon convergence to equilibrium in statistical physics, and
- Artificial Neural Networks – based upon features of biological neurons in terms of linear additivity for the inputs and strong nonlinearity for the resulting output.

As well as these general purpose optimization algorithms, there are also many problem-specific heuristics (such as the Karmarker-Karp procedure for number-partitioning).

It is not at all clear on which problem classes these algorithms are expected to perform well. If we are presented with a new problem demanding a solution we would like to be able to know a priori which of these algorithms is expected to perform best. To do this, we need a formal interpretation of their dynamics, motion in search space and convergence properties. From these we can derive a set of tests or summary of inherent advantages and disadvantages which will let to pick the best algorithm for a given purpose. Given that our search problem is difficult, it is unlikely that a first guess approach will yield sufficiently good answers (for whatever purpose we had in mind), so we cannot be happy with that status quo. Somehow the structure of

Random Search (Σ)

1. Pick a random solution $\sigma \in \Sigma$
2. **repeat**
3. Pick a random solution $\sigma' \in \Sigma$
4. If FITNESS(σ') $>$ FITNESS(σ)
5. then $\sigma \leftarrow \sigma'$
6. until No time remains
7. return σ

Figure 1.1: Random Search Algorithm

the search space (the almost infinite set of possible perturbation operators and the generated fitness landscapes) is is vital in dictating how the space should be searched efficiently.

In more direct terms, we must think about a process for generating or transforming solutions in a manner which we intuitively feel takes into account the deep structure of the problem. In short, without an understanding of the manner in which optimization algorithms operate, for every new case we need to invent a new heuristic, or rule of thumb, which seems particularly relevant. Clearly, this poses great difficulties for the unfortunate algorithm designer.

1.7.1 Random Search

If we don't have enough time to search all possible solutions, we could just generate solutions randomly for the length of time available, and retain the best one as our result. We formalize this as follows(see Figure 3.1):

It is quite clear that unless good solutions are pretty dense in the search space, a random search will almost always not perform well. However, another use of random search is as a benchmark — a standard level from which to compare other algorithms and with which to produce a very simple measure of how difficult a problem is (such comparisons will never be ideal, however, for at least the reason that different computer architectures and compilers may have different levels of performance across different types of computer code).

Choices for termination condition in the random search include time or solution quality. However since we cannot expect solutions of very good quality, the only practical condition is on the allowable search time.

1.7.2 Hill Climbing

A simple modification of the previous strategy is to pick solutions randomly and then examine their neighbourhood in the search space. The basic procedure is as follows – see Figure 3.2 (variants do exist):

This algorithm is called hill-climbing, because the inner loop (steps 4-8) move uphill (not necessarily in the steepest direction in the basic algorithm) until a summit is reached. The outer loop (steps 2-11) mean that the algorithm tries to find further hills if given enough time, and finally returns the highest peak it has visited when time runs out (this algorithm is sometimes called 'iterated hill climbing' because of the existence of the outer loop).

To use this algorithm we obviously require that the perturbation function be defined.

Whether hill-climbing works well as an optimization algorithm depends upon the local† structure of the landscape — if high points in the landscape are surrounded by a smooth slopes leading away long distances in the space, then any points picked on those slopes will lead to the top of the hill. The set of all such points is known as the 'attracting set', or 'capture basin' of the peak of the hill. If, on the other hand, tall peaks are surrounded by great crevasses then hill climbing will perform poorly. This image in terms of fitness landscape is extremely important in trying to understand when and why a particular algorithm will perform.

† Local being defined by the perturbation operator, of course.

Hill Climbing (Σ)

1. Pick a random solution $\sigma_{best} \in \Sigma$
2. repeat
3. Pick a random solution $\sigma \in \Sigma$
4. repeat
5. σ' $\leftarrow$ Perturbation(σ)
6. if FITNESS(σ') > FITNESS(σ)
7. then $\sigma \leftarrow \sigma$'
8. until No fitter perturbations of σ exist
9. if FITNESS(σ) > FITNESS(σ_{best})
10. then $\sigma_{best} \leftarrow \sigma$
11. until No time remains
12. return σ_{best}

Figure 1.2: Hill Climbing Algorithm

1.7.3 Genetic Algorithms

The idea of GAs developed by Holland (1975, 1992) as a computational method, based upon darwinian selection, of evoloving good solutions to a problem. See Figure 3.3.

The inruitive idea is simple: use a population of solutions and let them reproduce according to a process of selection and recombination (sexual reproduction). In the above implementation, we select a sub-population S from the entire population P (perhaps biasing the selction to those members of the population which are fitter), and then generate a new-population (all the old members die) via sexual reproduction between pairs of individuals from S, using the 'Crossover' function (usually mutation

Genetic Algorithm (Σ)

1. Pick a population $P = \{\sigma_1, \ldots, \sigma_n\}$ drawn randomly from Σ
2. repeat
3.   Form a selection:
4. $S \leftarrow$ Selection(P, Fitness())
5. $P \leftarrow \emptyset$
6.   repeat
7.   Pick a pair σ_a,σ_b from S
8.   Form the offspring:
9. $\sigma_{ab,1}$,$\sigma_{ab,2} \leftarrow$ Crossover(σ_a,σ_b)
10. $P \leftarrow P \cup \{\sigma_{ab,1},\sigma_{ab,2}\}$
11.   until Population is full: $|P| = n$
12. until $P \approx$ converged to a single solution
13. return $\sigma =$ arg $max_{\sigma \in \Sigma}$ {Fitness(σ)}

Figure 1.3: Genetic Algorithm

is also applied at a very low background rate). To be able to do this, we must be able to represent any solution as a collection of genes with particular values. This is always a huge choice for the manner in which this *representation* is constructed.

The ability of the algorithm to generate good solutions depend almost completely upon the representation chosen. A poor representation will result in the search being effectively random (or maybe even worse, guiding the search to poor solutions!), whereas an appropriate representation may produce high performance.

Despite a huge literature on genetic algorithms and their many variants, there is no real rigorous theory describing when such algorithms will perform well and when they will perform poorly. Their motivation is the belief natural evolution has been extraordinarily successful[‡], and the 'schema theorem' whic is of debatable relevance.

From the prespective of fitness landscapes, the crossover operator will only perform well on those landscapes which are highly correlated under its operation, but most common problems do not seem to fall within that category. This, amongst other ideas has lead certain researchers to suggest that the *co-evolution* of many different species generates landscapes which are effectively searched by crossover, and that we must understand the processes by which individual landscapes may be coupled and deform together.

1.7.4 Tabu Search

Tabu search is another general local search procedure. Though some of the ideas in tabu search were proposed over a period of time, tabu search was introduced in its present form by Glover [1989, 1990] and has since been applied to solve combinatorial optimization problems arising in many different areas [Glover et al., 1993].

Unlike simulated annealing, which is a memoryless search method, tabu search accumulates information during the search process and stores the information in memory. This information is used either to encourage or discourage the generation of certain types of solutions. Like simulated annealing, tabu search may accept a solution inferior to the current one.

Though accepting inferior solutions is necessary for local search methods to jump out of local optima, it may lead to cycling if the search repeatedly returns to a local optimum previously visited. The most important feature of tabu search is to use information stored in memory to forbid the generation of certain types of solutions. The conditions that characterize the forbidden solutions are called *tabu conditions*. Yet a solution that violates one or more of the tabu conditions can be accepted if

[‡]However, concepts such as 'optimal' are of lesser importance in evolutionary terms; often speed of adaptation and versatility are of greater importance. Suffice it to say that the evolution, need for, and ability of sexual reproduction and crossover are tricky issues still under active debate by biologists.

its aspiration value, computed according to an *aspiration function*, exceeds a certain value. Both the list of tabu conditions and the aspiration function are dynamically updated as the algorithm proceeds.

The main step in tabu search consists of generating the best solution in the neighborhood of the current solution subject to the tabu conditions (which can, however, be overriden according to the aspiration value of a solution), and replacing the current solution by the new solution. We now set up the general framework of tabu search. Let S denote the solution space and f the objective function of an optimization problem. Let $N(i)$ denote the neighborhood of solution i. Let T denote the tabu list and A the aspiration function. Figure 3.4 shows the general framework of tabu search.

Application of tabu search to an optimization problem has four key components: neighborhood structure, tabu list, aspiration function, and stopping condition; see Chapter 5 of [LS in CO] book.

1.7.5 Simulated Annealing

– covered in other chapter–

1.7.6 Artificial Neural Networks

– covered in other chapter–

begin TABU
  select an initial solution $i \in S$;
  initialize tabu list T and aspiration function A;
  while the *stopping condition is not met* **do**
    compute the best i' in $N(i)$
      subject to tabu list T and aspiration function A;
    $i \leftarrow i'$;
    update tabu list T;
    update aspiration function A;
  end while
end TABU

Figure 1.4: Tabu search paradigm

Chapter 2

Timetabling as a Generalized Assignment-Type Problem

The notion of the assignment-type problem and its generalized version appear to be powerful modeling tools. In particular, such models are very appropriate for formulating timetabling and scheduling problems where time periods have to be determined for activities according to several specific constraints.

2.1 Generalized Assignment-Type Problem

The Assignment-Type Problem (ATP) can be summarized as follows:

Given n items and m resources, the problem is to determine an assignment of each item to a resource in order to optimize an objective function and to satisfy K additional side constraints.

The mathematical model associated with an ATP is formulated as follows:

(ATP)

$$\text{Min} \quad F(x) \tag{2.1}$$

subject to

$$G_k(x) \leq 0 \qquad 1 \leq k \leq K \tag{2.2}$$

and

$$x \in X(1) \tag{2.3}$$

where

$$X(1) = \{x : \sum_{j \in J_i} x_{ij} = 1, 1 \leq i \leq n; xij = 0 \quad \text{or} \quad 1,\ 1 \leq i \leq b, j \in J_i\} \tag{2.4}$$

and $J_i \subset \{1, 2, \ldots, m\}$ is the set of admissible resources for item i, $1 \leq i \leq n$.

The decision variables x_{ij} are such that:

$$x_{ij} = \begin{cases} 1 & \text{if item } i \text{ assigned to resource } j \\ 0 & \text{otherwise.} \end{cases} \tag{2.5}$$

The objective function F and the side constraints G_k, $1 \leq k \leq K$, are only required to be calculable.

A Generalized Assignment-Type Problem (GATP) is an ATP where each item i has to be assigned to a_i resources, $a_i \geq 1$. The mathematical model associated with a GATP is:

(GATP)

$$\text{Min} \quad F(x) \tag{2.6}$$

subject to

$$G_k(x) \leq 0 \qquad 1 \leq k \leq K \tag{2.7}$$

and

$$x \in X(a) \tag{2.8}$$

where

$$X(a) = \{x : \sum_{j \in J_i} x_{ij} = a_i, 1 \leq i \leq n; xij = 0 \quad \text{or} \quad 1,\ 1 \leq i \leq n, j \in J_i\} \tag{2.9}$$

and

$$a = [a_1, a_2, \ldots, a_n]^T \tag{2.10}$$

and $a_1 \geq 1$ and integer $1 \leq i \leq n$. It is easy to see that an ATP is a GATP where $a = [1, 1, \ldots, 1]^T$.

The constraints $x \in X(a)$ underly several timetabling and scheduling problems. Furthermore, to take advantage of the simple structure of the constraints $x \in X(a)$, a *penalty approach* is used to deal with GATP. In this approach, let $V_k(x)$ denote the violation of constraint $G_k(x) \leq 0$; i.e.

$$V_k(x) = \text{ Max } \{0, G_k(x)\}. \tag{2.11}$$

Hence, for each $x \in X(a)$, let $V(x)$ denote the *total weighted violation*:

$$V(x) = \sum_{k=1}^{K} \beta_k V_k(x) \tag{2.12}$$

where $\beta_k \geq 0$ is proportional to the relative importance of the kth constraint. Then, instead of solving GATP, the solution techniques already introduced, are applied to the following *penalized version* of GATP:

(PGATP)

$$Min_{x \in X(a)} P(x) = \alpha F(x) + V(x) \tag{2.13}$$

where $\alpha \geq 0$. The values of α and β_k, $1 \leq k \leq K$, are parameters that can be adjusted according to the specific application. For instance, larger values for β_k, $1 \leq k \leq K$, allow putting more emphasis on feasibility.

2.2 Timetabling

Consider the problem of establishing the schedule of lectures accounting for individual student registrations, lecturers, and classroom availabilities. Note that this formulation allows for lectures of different length. Hence, to formulate this problem as a

GATP, the lectures are the items and the starting times allowed are the resources; then for each lecture i, $1 \leq i \leq n$, and starting time $j \in J_i$ (the set of admissible starting times for lecture i) such that $J_i = \{1, 2, \ldots, m\}$:

$$x_{ij} = \begin{cases} 1 & \text{if lecture } i \text{ starts at } j \\ 0 & \text{otherwise.} \end{cases} \tag{2.14}$$

The objective function accounts for the lecturer preferences:

$$F(x) = \sum_{i=1}^{n} \sum_{j \in J_i} c_{ij} x_{ij} \tag{2.15}$$

where c_{ij} is the cost of starting lecture i at j specified in terms of lecturer preferences ($c_{ij} = 0$ if j is the most preferred starting time for lecture i).

A first set of side constraints is specified to eliminate the conflicting situations where students or lecturers are involved in lectures taking place simultaneously. Define:

$\Gamma_{ij} = \{(k, l)$: lectures i and k have students or lecturers in common, and they overlap in time if lecture i starts at j and lecture k at l $\}$.

The conflicting constraints are specified as follows:

$$x_{ij} x_{kl} \leq 0 \qquad (k, l) \in \Gamma_{ij} \ , \ 1 \leq i \leq n \ , \ j \in J_i \ . \tag{2.16}$$

Other side constraints are introduced to account for classroom availability. Partition the set of classrooms available into several subsets including classrooms of the same type. Define:

B = number of classroom types U_b = number of classrooms of type b , $1 \leq b \leq B$.

For each lecture, a unique classroom type is specified. Define:

$K_{bt} = \{(i, j)$: lecture i requires a classroom of type b and it takes place during teaching hour t of the week whenever its starting time is j $\}$.

The classroom availability constraints are as follow:

$$\sum_{(i,j) \in K_{bt}} x_{ij} - U_b \leq 0 \qquad 1 \leq b \leq B \ , \ 1 \leq t \leq T \tag{2.17}$$

where T is the total number of teaching hours of the week.

The course timetabling problem can be summarized as follows:

$$Min \sum_{i=1}^{n} \sum_{j \in J_i} c_{ij} x_{ij} \ . \tag{2.18}$$

Subject to:

$$x_{ij} x_{kl} \leq 0 \qquad (k,l) \in \Gamma_{ij}, \quad 1 \leq i \leq n \ , \ j \in J_i \tag{2.19}$$

Next, we need to derive the associated penalized version of this problem (i.e. the penalized GATP), with weights specified to indicate the relative importance of violating the corresponding constraints. For the conflicting constraints, the weights account for the length of the conflict and the number of individuals involved. Define:

- L_{ijkl} = length (in number of hours) of the conflict $(k,l) \in \Gamma_{ij}$, $1 \leq i \leq n$, $j \in J_i$.
- δ_{ik} = number of students taking both lectures i and k, $1 \leq i \leq n$, $1 \leq k \leq n$.
- $\gamma_{ik} = \begin{cases} M + \delta_{ik} & \text{if lectures } i \text{ and } k \text{ have lecturers in common} \\ \delta_{ik} & \text{otherwise} \end{cases}$

  For $1 \leq i \leq n, 1 \leq k \leq n$.

where M is a parameter to indicate the relative importance of conflicts due to lecturers. Let $V_{kl}^{ij}(x)$ denote the violation of the conflicting constraints:

$$V_{kl}^{ij}(x) = Max\{0, x_{ij} x_{kl}\} = x_{ij} x_{kl} \ . \tag{2.20}$$

Let $V_{bt}(x)$ denote the violation of the classroom availability constraints:

$$V_{bt}(x) = Max\{0, \sum_{(i,j) \in K_{bt}} x_{ij} - U_b\} \ . \tag{2.21}$$

Furthermore, let the weight β_{kl}^{ij} associated with $V_{kl}^{ij}(x)$ be specified as follows:

$$\beta_{kl}^{ij} = \gamma_{ik} L_{ijkl} \quad . \tag{2.22}$$

Also, let the weight β_{bt} associated with $V_{bt}(x)$ be equal to a scalar ρ for all $1 \leq b \leq B$, $1 \leq t \leq T$. Then, the penalized GATP for this problem is as follows:

$$Min_{x \in X(1)} \sum_{i=1}^{n} \sum_{j \in J_i} \left[\alpha c_{ij} x_{ij} + \sum_{(k,l) \in \Gamma_{ij}} \gamma_{ik} L_{ijkl} x_{ij} x_{kl} \right] + \rho \sum_{b=1}^{B} \sum_{t=1}^{T} Max \left\{ 0, \sum_{(i,j) \in K_{bt}} x_{ij} - U_b \right\} \quad . \tag{2.23}$$

The weight parameters α, ρ, and M are selected according to the specific context. For instance, if the priority is to reduce the conflicts, then α and ρ take smaller values.

2.3 Testing the bibliography

Textbooks that cover a broad range of topics in combinatorial optimization include [Fou84, Law76, NW88, PR88, PS82, SM89, Sch87]. The book by Papadimitriou and Steiglitz is a good choice for a first course in combinatorial optimization. Unfortunately, it is currently out of print. The book by Nemhauser and Wolsey is a good choice for more advanced students, but I wouldn't use it for a first course in combinatorial optimization. I also like the book by Parker and Rardin, but its coverage of some topics is limited.

Computational Complexity theory plays an important role in combinatorial optimization, separating the easy problems that can be solved in polynomial time from the hard (particularly "NP-Hard") problems for which no polynomial time algorithms are known. Books on this topic include [GJ79, Pap94].

The first combinatorial optimization problems to be extensively studied were problems on graphs, including the shortest path problem, max flow problems, and matching. Books on this topic include [AMO93, Ber91, GKP92, Law76].

Many combinatorial optimization problems can be formulated as integer programming problems. In formulating an integer programming problem, there are many tricks that can be used to make the problem easier to solve. Books and articles on the formulation of integer programming and mixed integer programming problems include [AD92, BJN+93, Wil85].

Branch and bound algorithms are commonly used to solve integer programming problems. Branch and bound algorithms are discussed in nearly all of the textbooks described above, including [NW88, PR88, PS82, SM89]. Branch and bound and dynamic programming algorithms for the knapsack problem are discussed in [MT90].

Cutting plane algorithms are another approach to integer programming problems. These are described in [NW88, PR88, PS82, SM89]. Early cutting plane algorithms such as Gomory's cutting plane algorithm have not proven to be very effective. However, recent work in problem specific cutting plane algorithms and algorithms that combine branch and bound with cutting planes has been somewhat more successful. Some recent examples of cutting plane techniques include [Jun85, Law85, PR91, Sto92]. Some theoretical material on cutting planes can be found in [GLS88, Sch87]. Heuristics for combinatorial optimization problems are algorithms that find "good" solutions without any guarantee that the solutions are optimal. In recent years, there has been a lot of interest in heuristics, including genetic algorithms [Dav87, Dav91, Gol89], simulated annealing [AK89, AL87, Dav87, Ree93], and tabu search [Glo90b, Glo89, Glo90a, Ree93].

Approximation schemes are algorithms that are guaranteed to produce a solution that is within ϵ of optimality. Of particular interest are polynomial time approximation schemes- algorithms that given ϵ find an answer within a factor ϵ of optimality while running in polynomial time. Some approximation schemes for NP-hard problems are discussed in [PS82, PR88, HS84]. Unfortunately, there are also many NP-hard problems for which such approximation schemes are impossible unless P=NP. Some negative results are discussed in [PS82, PR88]. Very recently, work on transparent proofs has resulted in a large number of theorems about the non existence of polynomial time approximation schemes for various problems. These are discussed briefly in [Bab92, Joh92].

Algorithms and data structures are particularly important in implementing techniques for combinatorial optimization. Backtracking and dynamic programming are among the important algorithmic issues. Good implementations of data structures such as priority queues are important in getting good performance from the algorithms. Books that discuss these issues include [CLR90, HS84, Hu82, Tar83].

Scheduling problems are an important area of research in combinatorial optimization. Some books on this topic include [Bla93, Cof76, Hu82].

Matroids are a kind of algebraic structure within which some combinatorial optimization problems can easily be solved. Matroids are discussed in [Law76, Oxl92, NW88, PS82, PR88]. Greedoids, an extension of matroids, are described in [KLS91].

One interesting approach to integer programming problems is basis reduction. This is discussed in [GLS88, NW88, SM89, Sch87].

Duality in integer programming is another of research. This topic is discussed in [Fis81, NW88, PR88, Sha79].

There are a number of collection of test problems in combinatorial optimization that can be obtained via anonymous FTP. These include [Bea90, BBI92, Rei91].

Finally, a very useful, but slightly out of date annotated bibliography can be found in [OLK85].

Chapter 3

Algorithms and Heuristics

3.1 Summary

In this dissertation we try to answer the following question: How do we go about solving optimization problems, in general, and academic course scheduling (or timetabling) at the university level, in particular? we give a detailed answer to the latter part, and shed some light on the case of tackling optimization problems in general.

To summarize, we have investigated a variety of optimization algorithms, in particular, random search, hill climbing, knowledge-based systems (or expert systems), tabu search and simulated annealing-based methods to solve the course scheduling problem. The simulated annealing-based approach includes mean-field annealing, simulated annealing with three different cooling schedules, and the use of a rule-based preprocessor to provide a good initial solution for annealing.

The best results were obtained using simulated annealing with adaptive cooling and reheating as a function of cost, and a rule-based preprocessor. This approach enabled us to obtain valid schedules for the timetabling problem for a large university, using a complex cost function that includes student preferences. None of the other methods were able to provide a complete valid schedule.

3.2 Introduction

The primary objective of this dissertation is to shed light on how we go about solving optimization problems in general, and to present detailed analysis on deriving an approximate solution to the problem of university course scheduling in particular.

In order to solve an optimization problem such as the TSP, we design a series of steps, or a procedure, which we can problem instance to which it claims to adhere and, ideally, be sure of getting a good answer in a given amount of time*. Such a procedure is known as an 'optimization algorithm'.

Sometimes we may be happy with (or forced to accept) a less than optimal solution, in which case it may be easier to find a suitable algorithm. We'd like our algorithms to return as good an answer as possible in as little time as possible, and these will be important comparative judges of the merit of different algorithms. Again, this is more subtle than it seems in that the time will vary from problem instance to another, and we have to decide whether we want to minimize the best case, the worst case or the average case. The reason NP-complete problems are considered difficult is that all known algorithms which guarantee the optimal solution will operate in exponential time in the size of the problem in the worst case scenario. For reasonably large problem instances this is impractical and so we reduce our expectations of the solution from 'optimal' to 'good', and seek algorithms which can still search effectively in these difficult search spaces. As well as average versus best/worst case in time-requirements, we also want to understand average versus best/worst case in quality of solution found.

The optimization problem we have closely tackled is university course scheduling (or timetabling), which can be summarized as follows:

given data sets of classes and their days, enrollments, and instructors; rooms and their capacities, types, and locations; distances between buildings; priorities of each building for different departments; and students and their class preferences; the

*Usually we can avoid complications due to the Halting problem, as we address problems of a simpler nature in which we are really fighting against NP-completeness and are sure that a solution does exist.

problem is to construct a feasible class schedule satisfying all the hard constraints and minimizing the medium and soft constraints. Hard constraints are space and time constraints that must be satisfied, such as scheduling only one class at a time for any teacher, student, or classroom. Medium and soft constraints are student and teacher preferences that should be satisfied if possible.

The timetabling problem (TTP) is a high-dimensional, non-Euclidean, multi-constraint combinatorial optimization problem, and is consequently very difficult to solve. This problem has been tackled by many researchers, mostly in the field of operations research. A number of different heuristics have been tried on different instances of the problem, from high school to university course scheduling (see the reviews by de Werra [?] and Shaerf [?] and the papers collected in Ref. [?]). For small to medium size problems, such as exam scheduling, high school scheduling, or course scheduling for a university department, many of these methods work well. However no particular method has yet been shown to produce good results for real-world problems on a much larger scale, such as scheduling all courses for a large university, which we address in this paper. Also, we are not aware of any large scale study that takes into account constraints due to student preferences, as we have done.

We have used data for classes at Syracuse University. Currently this problem is handled by the university scheduling department in a semi-automated fashion. A scheduling program is used to find a partial solution, and substantial manual effort is required to iterate towards a final solution. Also, when scheduling a certain semester (e.g. fall 1996), a template of a previous semester (e.g. fall 1995) is used as part of the input data. We have applied the following optimization techniques to this problem:

1. A rule-based expert system.
2. Mean-field annealing.
3. Simulated annealing with geometric cooling.
4. Simulated annealing with adaptive cooling.
5. Simulated annealing with adaptive cooling and reheating as a function of cost.

6. Simulated annealing (using each of the three different cooling schedules) with a rule-based preprocessor to provide a good initial solution.

The best results were obtained using simulated annealing with adaptive cooling and reheating as a function of cost, and with a rule-based preprocessor to provide a good initial solution. Using this method, and with careful selection of parameters and update moves, we were able to generate solutions to the class scheduling problem using real data for a large university. None of the other methods were able to provide a complete valid solution.

3.3 Approximation Algorithms

When facing an *NP*-complete optimization, we may want to consider algorithms that do not produce optimum solutions, but solutions *guaranteed to ve close to the optimum.* Suppose that we wish to obtain such solutions for an optimization problem, maximization or minimization. For each instance x of this problem, there is an optimum solution with value $opt(x)$; let us assume that $opt(x)$ is always a positive integer.

Suppose now that we have a polynomial algorithm A which, when presented with instance x of the optimization problem, returns some solution with value $A(x)$. Since the problem is *NP*-complete and A is polynomial, we cannot realistically hope that $A(x)$ is always the optimum value. But suppose that we know that the following inequality hold:

$$\frac{|opt(x) - A(x)|}{opt(x)} \leq \epsilon \tag{3.1}$$

where ϵ is some positive real number, hopefully very small, that bounds from above the worst-case relative error of algorithm A. (The absolute value in this inequality allows us to treat both minimization and maximization problems within the same framework.) If algorithm A satisfies this inequality for all instances x of the problem, then it is called an **ϵ-approximation algorithm.**

Once an optimization problem has been shown to be NP-complete, the following question becomes most important: Are there ϵ-approximation algorithms for this problem? And if so, how small can ϵ be? Let us observe at the outset that such questions are meaningful only if we assume that $p \neq NP$, because, if $P = NP$, then the problem can be solved exactly, with $\epsilon = 0$.

3.4 Type of problems addressed

A question to ask here is that: What problems can stochastic optimization algorithms such as simulated annealing address?

Most of the algorithms we have looked at adhere to a simple set of requirements upon the space of problems to which they apply. In fact we require no more of a problem than that:

1. we can generate a random solution,
2. a 'local perturbation' to a solution is defined,
3. such perturbations over all relevant parts of the solution space (this is usually naturally satisfied).

All of the algorithms discussed below fall within this category. Also note that because we search directly in the problem space, all the algorithms we consider will return both the actual solution and its cost (or value of utility depending upon the context). One can also envisage algorithms which only return a bound upon the solution cost, and not the actual solution. Whilst of theoretical value, in most practical instances we would ordinarily require the solution as well. Finally, it is important to note that:

- individual problem instances may vary enormously in difficulty. The average versus best or worse case difficulty (or complexity) of search and problem solving may depend in quite particular fashions on the method of problem selection. These issues were addressed later in greater depth.

- these algorithms may not specify a natural termination condition, and in such cases heuristic principles must be used. Possibilities include: maximum running time, solution quality, time since last improvement; or combinations thereof.

In the descriptions which follow, we shall assume we are supplied with a function 'FITNESS: $\sum \rightarrow R$' which supplies us with the solution quality. Without loss of generality, we assume higher fitnesses are better.

3.5 Heuristic and Algorithms

How in general should we go about designing an algorithm for a particular purpose? Looking at the above algorithms, we see they are derived from a wide array of fields of knowledge:

- Random Search – a no-knowledge approach;
- Hill Climbing – the most basic idea given the concept of a fitness landscape;
- Genetic Algorithms – based upon evolution and the power of recombination of partial solutions;
- Tabu Search – an extension of classical local improvement heuristics. The search is conducted in a search space made up of all possible solutions considered;
- Simulated Annealing – based upon convergence to equilibrium in statistical physics, and
- Artificial Neural Networks – based upon features of biological neurons in terms of linear additivity for the inputs and strong nonlinearity for the resulting output.

As well as these general purpose optimization algorithms, there are also many problem-specific heuristics (such as the Karmarker-Karp procedure for number-partitioning).

It is not at all clear on which problem classes these algorithms are expected to perform well. If we are presented with a new problem demanding a solution we would like to be able to know a priori which of these algorithms is expected to perform best. To do this, we need a formal interpretation of their dynamics, motion in search space and convergence properties. From these we can derive a set of tests or summary of inherent advantages and disadvantages which will let to pick the best algorithm for a given purpose. Given that our search problem is difficult, it is unlikely that a first guess approach will yield sufficiently good answers (for whatever purpose we had in mind), so we cannot be happy with that status quo. Somehow the structure of the search space (the almost infinite set of possible perturbation operators and the generated fitness landscapes) is is vital in dictating how the space should be searched efficiently.

In more direct terms, we must think about a process for generating or transforming solutions in a manner which we intuitively feel takes into account the deep structure of the problem. In short, without an understanding of the manner in which optimization algorithms operate, for every new case we need to invent a new heuristic, or rule of thumb, which seems particularly relevant. Clearly, this poses great difficulties for the unfortunate algorithm designer.

3.5.1 Random Search

If we don't have enough time to search all possible solutions, we could just generate solutions randomly for the length of time available, and retain the best one as our result. We formalize this as follows(see Figure 3.1):

It is quite clear that unless good solutions are pretty dense in the search space, a random search will almost always not perform well. However, another use of random search is as a benchmark — a standard level from which to compare other algorithms and with which to produce a very simple measure of how difficult a problem is (such comparisons will never be ideal, however, for at least the reason that different computer architectures and compilers may have different levels of performance across

Random Search (Σ)

1. Pick a random solution $\sigma \in \Sigma$
2. **repeat**
3.   Pick a random solution $\sigma' \in \Sigma$
4.   If FITNESS(σ') > FITNESS(σ)
5.   then $\sigma \leftarrow \sigma'$
6. until No time remains
7. return σ

Figure 3.1: Random Search Algorithm

different types of computer code).

Choices for termination condition in the random search include time or solution quality. However since we cannot expect solutions of very good quality, the only practical condition is on the allowable search time.

3.5.2 Hill Climbing

A simple modification of the previous strategy is to pick solutions randomly and then examine their neighbourhood in the search space. The basic procedure is as follows – see Figure 3.2 (variants do exist):

This algorithm is called hill-climbing, because the inner loop (steps 4-8) move uphill (not necessarily in the steepest direction in the basic algorithm) until a summit is reached. The outer loop (steps 2-11) mean that the algorithm tries to find further hills if given enough time, and finally returns the highest peak it has visited when time runs out (this algorithm is sometimes called 'iterated hill climbing' because of the existence of the outer loop).

To use this algorithm we obviously require that the perturbation function be defined.

Hill Climbing (Σ)

Pick a random solution $\sigma_{best} \in \Sigma$

repeat

  Pick a random solution $\sigma \in \Sigma$

  repeat

    σ' $\leftarrow$ Perturbation(σ)

    if FITNESS(σ') $>$ FITNESS(σ)

    then $\sigma \leftarrow \sigma$'

  until No fitter perturbations of σ exist

  if FITNESS(σ) $>$ FITNESS(σ_{best})

  then $\sigma_{best} \leftarrow \sigma$

until No time remains

return σ_{best}

Figure 3.2: Hill Climbing Algorithm

Whether hill-climbing works well as an optimization algorithm depends upon the local[†] structure of the landscape — if high points in the landscape are surrounded by a smooth slopes leading away long distances in the space, then any points picked on those slopes will lead to the top of the hill. The set of all such points is known as the 'attracting set', or 'capture basin' of the peak of the hill. If, on the other hand, tall peaks are surrounded by great crevasses then hill climbing will perform poorly. This image in terms of fitness landscape is extremely important in trying to understand when and why a particular algorithm will perform.

[†] Local being defined by the perturbation operator, of course.

Genetic Algorithm (Σ)

1. Pick a population $P = \{\sigma_1, \ldots, \sigma_n\}$ drawn randomly from Σ
2. repeat
3. Form a selection:
4. $S \leftarrow$ Selection(P, Fitness())
5. $P \leftarrow \emptyset$
6. repeat
7. Pick a pair σ_a,σ_b from S
8. Form the offspring:
9. $\sigma_{ab,1}$,$\sigma_{ab,2} \leftarrow$ Crossover(σ_a,σ_b)
10. $P \leftarrow P \cup \{\sigma_{ab,1},\sigma_{ab,2}\}$
11. until Population is full: $|P| = n$
12. until $P \approx$ converged to a single solution
13. return $\sigma =$ arg $max_{\sigma\in\Sigma}$ $\{$Fitness(σ)$\}$

Figure 3.3: Genetic Algorithm

3.5.3 Genetic Algorithms

The idea of GAs developed by Holland (1975, 1992) as a computational method, based upon darwinian selection, of evoloving good solutions to a problem. See Figure 3.3.

The inruitive idea is simple: use a population of solutions and let them reproduce according to a process of selection and recombination (sexual reproduction). In the above implementation, we select a sub-population S from the entire population P (perhaps biasing the selction to those members of the population which are fitter), and then generate a new-population (all the old members die) via sexual reproduction between pairs of individuals from S, using the 'Crossover' function (usually mutation is also applied at a very low background rate). To be able to do this, we must be able to represent any solution as a collection of genes with particular values. This is always a huge choice for the manner in which this *representation* is constructed.

The ability of the algorithm to generate good solutions depend almost completely upon the representation chosen. A poor representation will result in the search being effectively random (or maybe even worse, guiding the search to poor solutions!), whereas an appropriate representation may produce high performance.

Despite a huge literature on genetic algorithms and their many variants, there is no real rigorous theory describing when such algorithms will perform well and when they will perform poorly. Their motivation is the belief natural evolution has been extraordinarily successful[‡], and the 'schema theorem' whic is of debatable relevance.

From the prespective of fitness landscapes, the crossover operator will only perform well on those landscapes which are highly correlated under its operation, but most common problems do not seem to fall within that category. This, amongst other ideas has lead certain researchers to suggest that the *co-evolution* of many different species generates landscapes which are effectively searched by crossover, and that we must understand the processes by which individual landscapes may be coupled and deform together.

3.5.4 Tabu Search

Tabu search is another general local search procedure. Though some of the ideas in tabu search were proposed over a period of time, tabu search was introduced in its present form by Glover [1989, 1990] and has since been applied to solve combinatorial optimization problems arising in many different areas [Glover et al., 1993].

[‡]However, concepts such as 'optimal' are of lesser importance in evolutionary terms; often speed of adaptation and versatility are of greater importance. Suffice it to say that the evolution, need for, and ability of sexual reproduction and crossover are tricky issues still under active debate by biologists.

Unlike simulated annealing, which is a memoryless search method, tabu search accumulates information during the search process and stores the information in memory. This information is used either to encourage or discourage the generation of certain types of solutions. Like simulated annealing, tabu search may accept a solution inferior to the current one.

Though accepting inferior solutions is necessary for local search methods to jump out of local optima, it may lead to cycling if the search repeatedly returns to a local optimum previously visited. The most important feature of tabu search is to use information stored in memory to forbid the generation of certain types of solutions. The conditions that characterize the forbidden solutions are called *tabu conditions*. Yet a solution that violates one or more of the tabu conditions can be accepted if its aspiration value, computed according to an *aspiration function*, exceeds a certain value. Both the list of tabu conditions and the aspiration function are dynamically updated as the algorithm proceeds.

The main step in tabu search consists of generating the best solution in the neighborhood of the current solution subject to the tabu conditions (which can, however, be overriden according to the aspiration value of a solution), and replacing the current solution by the new solution. We now set up the general framework of tabu search. Let S denote the solution space and f the objective function of an optimization problem. Let $N(i)$ denote the neighborhood of solution i. Let T denote the tabu list and A the aspiration function. Figure 3.4 shows the general framework of tabu search.

Application of tabu search to an optimization problem has four key components: neighborhood structure, tabu list, aspiration function, and stopping condition; see Chapter 5 of [LS in CO] book.

3.5.5 Simulated Annealing

– see section below –

```
begin TABU
  select an initial solution i ∈ S;
  initialize tabu list T and aspiration function A;
  while the stopping condition is not met do
    compute the best i' in N(i)
      subject to tabu list T and aspiration function A;
    i ← i';
    update tabu list T;
    update aspiration function A;
  end while
end TABU
```

Figure 3.4: Tabu search paradigm

3.5.6 Artificial Neural Networks

– see mean field annealing below —

3.6 The Timetabling Problem

Timetabling is the assignment of time slots to a set of events, subject to constraints on these assignments. The NP-complete *professors and classes* timetabling problem [**?**, **?**, **?**] is a constraint satisfaction problem that can be briefly stated as follows:

For a certain school with N_p professors, N_q classes, N_x classrooms and lecture halls, and N_s students, it is required to schedule N_l professor-class pairs within a time limit of N_t time slots producing a legal schedule. A legal schedule needs to be found such that no professor, class, or student is in more than one place at a time, and no room is expected to accommodate more than one lesson at a time or more students than its capacity.

The constraints for this problem can be hard, medium or soft. The medium and soft constraints have an associated cost (or penalty), and if they are not satisfied, the goal is to minimize this cost. Soft constraints have a lower priority (and thus lower cost) than medium constraints. The hard constraints must be satisfied, so their associated cost must be reduced to zero. A feasible schedule is one that satisfies all the hard constraints.

Hard constraints are usually constraints that physically cannot be violated. This includes events that must not overlap in time, such as:

- classes taught by the same professor,
- classes held in the same room,
- a class and a recitation or a lab of the same class.

Another examples are space or room constraints:

- A class cannot be assigned to a particular room unless the capacity of the room is greater than or equal to the class enrollment.
- Some classes, such as laboratories, require a certain type of room.

Medium constraints are usually considered to be those constraints that fall into the gray area between the hard and soft constraints [?]. In our implementation, we define medium constraints to be constraints such as time and space conflicts which, like hard constraints, cannot physically be violated (for example, it is not possible for one person to be in two different classes at the same time). However we consider these constraints to be medium rather than hard if they can be avoided by making adjustments to the specification of the problem. The primary example is student preferences. We cannot expect to be able to satisfy all student class preferences, in some cases, certain students will have to adjust their preferences since certain classes will clash, or will be oversubscribed.

Medium constraints have a high penalty attached to them, although not as high as that associated with the hard constraints. In the final schedule the penalty of these constraints should be minimized and preferably reduced to zero. Some examples of medium constraints are:

- Avoid time conflicts for classes with students in common.
- Eligibility criteria for the class must be met.
- Do not enroll athletes in classes that conflict with their sport practice time (of course, depending on the sport).

Soft constraints are preferences that do not deal with time conflicts, and have a lower penalty (or cost) associated with them. We aim to minimize the cost, but do not expect to be able to reduce it to zero. Some examples are:

- For each student, balance the three-day (*Mon, Wed, Fri*) as well as the two-day (*Tue, Thu*) schedules.
- Balance or spread out the lectures over the week.
- Classes may request contiguous time slots.
- Balance enrollment in multi-section classes.
- Lunch and other break times may be specified.
- Professors may request periods in which their classes are not taught.
- Professors may have preferences for specific rooms or types of rooms.
- Once a class enrollment exceeds the upper limit, the class is split into multi-sections.
- Minimize the distance between the room where the class is assigned and the building housing its home department.

Some soft constraints may have higher priority (and thus higher cost) than others. For example, preferences involving professors will have higher priority than the preferences of students.

Other constraints that may need to be taken into account during the scheduling process:

- It is not the case, in general, that all classrooms are available for all subjects. In reality, many subjects require special purpose rooms.
- Two course of the same topic are scheduled on two different working days, if possible.
- Some rooms may be declared unavailable during any time period.
- Daily limits must be specified to restrict professors from taking more than a specified number of classes per day.

The **cost function** measures the quality of the current schedule and generally involves the weighted sum of penalties associated with different types of constraint violations. The aim of the optimization technique is to minimize the cost function.

3.7 Mean-Field Annealing

One of the potential drawbacks of using simulated annealing for hard optimization problems is that finding a good solution can often take an unacceptably long time. Mean-field annealing (MFA) attempts to avoid this problem by using a deterministic approximation to simulated annealing, by attempting to average over the statistics of the annealing process. The result is improved execution speed at the expense of solution quality. Although not strictly a continuous descent technique, MFA is closely related to the Hopfield neural network [**?**, **?**].

Mean-field annealing has been successfully applied to high school class scheduling [**?**]. For scheduling, it is advantageous to use a Potts neural encoding to specify discrete neural variables (or neurons) for the problem. This is defined in its simplest form as a mapping of events onto space-time slots, for example an event i, in this case a professor-class pair (p, q), is mapped onto a space-time slot a, in this case a classroom-timeslot pair (x, t). Now, the Potts neurons S_{ia} are defined to be 1 if event i takes place in space-time slot a, and 0 otherwise. In this way, the constraints involved can be embedded in the neural net in terms of the weights $w_{i,j}$ of the neural network, which encode a Potts normalization condition such as $\sum_a S_{ia} = 1$.

For a full derivation of the mean-field annealing algorithm from its roots in statistical physics, see Hertz *et al.* [?] or Peterson *et al.* [?]. Here we will just give a brief overview of the method. The basic idea is that it is possible to approximate the actual cost or energy function E, which is a function of discrete neural variables S_{ia}, by an effective energy function $E^{'}$ that can be represented in terms of continuous variables U_{ia} and V_{ia}. These are known as mean field variables, since V_{ia} is an approximation to the average value of S_{ia} at a given temperature T.

This approach effectively smooths out the energy function and makes it easier to find the minimum value, which is obtained by solving the saddle point equations $\frac{\partial E^{'}}{\partial V_{ia}} = 0$ and $\frac{\partial E^{'}}{\partial U_{ia}} = 0$, which generate a set of self-consistent mean field theory (MFT) equations in terms of the mean field variables U and V:

$$U_{ia} = -\frac{1}{T}\frac{\partial E}{\partial V_{ia}} \tag{3.2}$$

$$V_{ia} = \frac{e^{U_{ia}}}{\sum_b e^{U_{ib}}}. \tag{3.3}$$

The MFA algorithm involves solving equations 4.1 and 4.2 at a series of progressively lower temperatures T: this process is known as temperature annealing. The critical temperature T_c, which sets the scale of T, is estimated by expanding equation 4.2 around the trivial fixed-point [?, ?] $V_{ia}^{(0)} = \frac{1}{N_a}$, where N_a is the number of possible states of each of the network neurons. For example, for the events defined by professor-class pairs (p, q) mapped onto classroom-timeslots (x, t), we have $N_p N_q$ neurons, each of which has $N_x N_t$ possible states, in which case $V_{pq;xt}^{(0)} = \frac{1}{N_x N_t}$.

Equations 4.1 and 4.2 can be solved iteratively using either synchronous or serial updating. The iterative dynamics to evolve the mean field variables toward a self-consistent solution is explained in detail by Peterson *et al.* [?]. The solutions correspond to stable states of the Hopfield network [?]. Observe from equation 4.2 that any solution to the MF equations respects a continuous version of the Potts condition

$$\sum_a V_{ia} = 1 \quad \forall\, i. \tag{3.4}$$

3.7.1 The Mean-Field Annealing Algorithm

The generic MFA algorithm appears in Figure 4.1. At high temperatures T, the mean-field solutions will be states near the fixed-point symmetrical maximum entropy state $V_{ia} = 1/N_a$. At low temperatures, finding a mean-field solution will be equivalent to using the Hopfield model, which is highly sensitive to the initial conditions and known to be ineffective for hard problems [?]. MFA improves over the Hopfield model by using annealing to slowly decrease the temperature in order to sidestep these problems.

These characteristics are similar to those of simulated annealing, which is no surprise since both it and the mean-field method compute thermal averages over Gibbs distributions of discrete states, the former stochastically and the latter through a deterministic approximation. It is therefore natural to couple the mean-field method with the concept of annealing from high to low temperatures.

In addition to the structure of the energy function, there are three major interdependent issues which arise in completely specifying a mean-field annealing algorithm for a timetabling problem:

- The values of the coefficients of terms in the energy function.
- The types of dynamics used to find solutions of the MFT equations at each T.
- The annealing schedule details, i.e. the initial temperature $T(0)$, the rules for deciding when to reduce T and by how much, and the termination criteria.

1. Choose a problem and encode the constraints into weights $\{w_{ij}\}$.

2. Find the approximate phase transition temperature by linearizing equation (4.2).

3. Add a self-coupling β-term if necessary. In a neural net, this corresponds to a feedback connection from a neuron to itself.

4. Initialize the neurons V_{ia} to high temperature values $\frac{1}{N_a}$ plus a small random term such as $rand[-1,1] \times 0.001$; and set $T(0) = T_c$.

5. Until $(\Sigma \geq 0.99)$ do:

   - At each $T(n)$, update all U_{ia} and V_{ia} by iterating to a solution of the mean field equations.
   - $T(n+1) = \alpha T(n)$, we chose $\alpha = 0.9$

6. The discrete values S_{ia} that specify the schedule are obtained by rounding the mean field values V_{ia} to the nearest integer (0 or 1).

7. Perform *greedy heuristics* if needed to account for possible imbalances or rule violations.

Figure 3.5: The Generic Mean-Field Annealing Algorithm

Peterson *et al.* [?] introduced a quantity called *saturation*, Σ, defined as

$$\Sigma = \frac{1}{N_i} \sum_{ia} V_{ia}^2 \ , \tag{3.5}$$

where N_i is the number of events (in this case the number of professor-class pairs). This characterizes the degree of clustering of events in time and/or space, $\Sigma_{min} = \frac{1}{N_a}$ corresponds to high temperature, whereas $\Sigma_{max} = 1$ means that all the V_{ia} have converged to 0 or 1 values, indicating that each event has been assigned to a space-time slot.

The first step of Figure 4.1 is to map the constraints of the problem into the neural net connection weights. In our implementation, at each T_n the MFA algorithm (Figure 4.1) performs one update per neural variable (defined as one sweep) with sequential updating using equations 4.1 and 4.2. After reaching a saturation value close to 1 (we chose $\Sigma = 0.99$) we check whether the obtained solutions are valid, i.e. $E_{hard} = 0$. If this is not the case the network is re-initialized and is allowed to resettle. We repeat this procedure a number of times until the best solution is found. A similar procedure was carried out on high school scheduling by Peterson *et al.* [?].

The MFA implementation was a little more complicated than the implementation of simulated annealing and the expert system, since it had many more parameters to handle, and it was often more difficult to find optimal values for these parameters. For example, one complication is the computation of the critical temperature T_c, which involved an iterative procedure of a linearized dynamic system. On the other hand, we observed that the convergence time was indeed much less than any of the convergence times of the simulated annealing using the three annealing schedules studied. For more details on our MFA implementation, see Ref. [?].

3.8 The Rule-Based System

We have implemented a fairly complex rule-based expert system for solving the timetabling problem, for three reasons. Firstly, it gives us a benchmark as to how well other methods do in comparison to this standard technique. Secondly, a simplified version of the rule-based system is used to provide sensible choices for moves in the simulated annealing algorithm, rather than choosing swaps completely at random, and this greatly improves the proportion of moves that are accepted. Thirdly, we have used this system as a preprocessor for simulated annealing, in order to provide a good initial solution.

Simulated annealing is a very time-consuming, computationally intensive procedure. Using an expert system as a preprocessor is a way of quickly providing a good starting point for the annealing algorithm, which reduces the time taken in the annealing procedure, and improves the quality of the result. Our results clearly support this rationale for the case of academic scheduling.

The rule-based expert system consists of a number of rules (or heuristics) and conventional recursion to assist in carrying out class assignments. We have developed this system specifically for the problem of academic scheduling. The basic data structures or components of the system are:

1. Distance matrix of values between each academic department and every other building under use for scheduling.

2. Class data structure of each class scheduled anywhere in campus. These structures are capable of linking with each other.

3. Room data structure of each room (regardless of type) involved in the scheduling process. Like classes, room structures are also linked with each other.

4. Data structures for time periods to keep track of which hour or time slot was occupied and which was not.

5. Department inclusion data structure giving department inclusion within other larger departments or colleges.

6. Students structures indicating classes of various degree of requirements and preferences for each student.

The basic function of the system is as follows: given data files of classes, rooms and buildings, department-to-building distance matrix, students data, and the inclusion data, using the abovementioned data structures, the system builds an internal database which in turn is used in carrying out the scheduling process. This process involves a number of essential sub-processes such as checking the distances between buildings, checking building, room type and hours occupied, checking and comparing time slots for any conflicts, checking rooms for any space conflict, and keeping track of and updating the hours already scheduled.

The rule-based system uses an iterative approach. The basic procedure for each iteration is as follows. The scheduling of classes is done by department, so each iteration consists of a loop over all departments. The departments are chosen in order of size, with those having the most classes being scheduled first. The system first loops over all the currently unscheduled classes, and attempts to assign them to the first unoccupied room and timeslot that satisfies all the rules governing the constraints. Since constraints involving capacity of rooms are very difficult to satisfy, larger classes are scheduled first, to try to avoid not having large enough rooms later for those class sections with large enrollments.

In some cases the only rooms and timeslots that satisfy all the rules will already be occupied by previously scheduled classes. In that case, the system attempts to move one of these classes into a free room and timeslot, to allow the unscheduled class to be scheduled.

Next, the system searches through all the scheduled classes, and selects those that have a high cost, by checking the medium and soft constraints such as how closely the room size matches the class size, how many students have time conflicts, whether the class is in a preferred time period or a preferred building, and so on. Selecting threshold values for defining what is considered a "high" cost in each case is a subjective procedure, but it is straightforward to choose reasonable values. When a poorly scheduled class is identified, the system searches for a class to swap it with, so that the hard constraints are still satisfied, but the overall cost of the medium and soft constraints is reduced.

This process of swapping rooms continues provided all the rules are satisfied and no "cycling" (swapping of the same classes) occurs. Once all the departments have been considered, this completes one iteration. The system continues to follow this iterative procedure until a complete iteration produces no changes to the schedule.

There are many rules dealing with space and hours, type of room, and priority of room. Many are quite complex, but some of the basic rules, such as those implementing the hard constraints, can be quite straightforward – for example, the following is the basic rule for dealing with time and space conflicts for a room:

IF [room(capacity) > class(space-requested)] and [no time conflict in this room] THEN assign the room to the class.

When the rule-based system is used as a preprocessor, it produces a partial schedule as an output, since it is usually unable to assign all of the given classes to rooms and times slots. The output is divided into two parts: the first consists of classes, with their associated professors and students, assigned to various rooms; and the second is a list of classes that could not be assigned due to constraint conflicts.

3.9 Simulated Annealing

Simulated annealing (SA) has been widely used for tackling different combinatorial optimization problems, particularly academic scheduling [**?**, **?**, **?**]. The basic algorithm is described in Figure 4.2. The results obtained depend heavily on the cooling schedule used. We initially used the most commonly known and used schedule, which is the geometric cooling, but later tried adaptive cooling, as well as the method of geometric reheating based on cost [**?**].

A comprehensive discussion of the theoretical and practical details of SA is given in [**?**, **?**, **?**, **?**]. It suffices here to say that the elementary operation in the Metropolis method for a combinatorial problem such as scheduling is the generation of some new candidate configuration, which is then automatically accepted if it lowers the cost (C), or accepted with probability $\exp(-\Delta C/T)$, where T is the temperature, if it would increase the cost by $\Delta(C)$. Also, in Figure 4.2, s is the current schedule and $s^{'}$ is a neighboring schedule obtained from the current neighborhood space ($\mathcal{N}_s$) by swapping two classes in time and/or space.

Thus the technique is essentially a generalization of the local optimization strategy, where, at non-zero temperatures, thermal excitations can facilitate escape from local minima.

The SA algorithm has advantages and disadvantages compared to other global optimization techniques. Among its advantages are the relative ease of implementation, the applicability to almost any combinatorial optimization problem, the ability to provide reasonably good solutions for most problems (depending on the cooling schedule and update moves used), and the ease with which it can be combined with

1. Generate an initial schedule s.
2. Set the initial best schedule $s^* = s$.
3. Compute cost of $s : C(s)$.
4. Compute initial temperature T_0.
5. Set the temperature $T = T_0$.
6. While *stop criterion* is not satisfied do:
   (a) Repeat *Markov chain length* (M) times:
      i. Select a random neighbor s' to the current schedule, $(s' \subset \mathcal{N}_s)$.
      ii. Set $\Delta(C) = C(s') - C(s)$.
      iii. If $(\Delta(C) \leq 0$ {downhill move}):
         - Set $s = s'$.
         - If $C(s) < C(s^*)$ then set $s^* = s$.
      iv. If $(\Delta(C) > 0$ {uphill move}):
         - Choose a random number r uniformly from $[0, 1]$.
         - If $r < e^{-\Delta(C)/T}$ then set $s = s'$.
   (b) Reduce (or update) temperature T.
7. Return the schedule s^*.

Figure 3.6: The Simulated Annealing Algorithm

other heuristics, such as expert systems, forming quite useful hybrid methods for tackling a range of complex problems. SA is a robust technique, however, it does have some drawbacks. To obtain good results the update moves and the various tunable parameters used (such as the cooling rate) need to be carefully chosen, the runs often require a great deal of computer time, and many runs may be required.

Depending on the problem to which it is applied, SA appears competitive with many of the best heuristics, as shown in the work of Johnson *et al.* [?].

3.9.1 Timetabling Using the Annealing Algorithm

The most obvious mapping of the timetabling problem into the SA algorithm involves the following constructs:

1. a **state** is a timetable containing the following sets:
   - P: a set of professors.
   - C: a set of classes.
   - S: a set of students.
   - R: a set of classrooms.
   - I: a set of time intervals.
2. a **cost** or "energy" E(P, C, S, R, I) such that:
   - E(P): is the cost of assigning more than maximum number of allowed classes M_p to the same professor, plus scheduling one or more classes that cause a conflict in the professor's schedule.
   - E(C): is the cost of scheduling certain classes at/within the same time period in violation of the exclusion constraint, for example.
   - E(S): is the cost of having two or more classes conflict in time; plus cost of having in the schedule one or more classes that really don't meet the student's major, class requested, or class requirements; plus the cost of not having the classes evenly spread out over the week, etc.
   - E(R): is the cost resulting from assigning room(s) of the wrong size and/or type to a certain class.
   - E(I): is the cost of having more or less time periods than required, plus cost of an imbalanced class assignments (a certain period will have more classes assigned to than others, etc.).

3. A **swap** (or a move) is the exchange of one or more of the following: class c_i with class c_j in the set C with respect to time periods I_i and I_j, and/or with respect to classroom R_i and R_j, respectively. Generally, this step is referred to as class swapping.

Along with all of the necessary constraints, the simulated annealing algorithm also takes as input data the following: the preprocessor output in the form of lists of scheduled and non-scheduled classes and their associated professors and room types, a list of rooms provided by the registrar's office, a department to building distance matrix, a list of students and their class preferences, and a list of classes that are not allowed to be scheduled simultaneously.

To use simulated annealing effectively, it is crucial to use a good cooling schedule, and a good method for choosing new trial schedules, in order to efficiently sample the search space. We have experimented with both these areas, which are discussed in the following sections.

3.9.2 The Annealing Schedules

Three annealing schedules have been used in our experiments to update the temperature of the SA algorithm in Figure 4.2: geometric cooling, adaptive cooling, and adaptive reheating as a function of cost.

The first schedule we have used is **geometric cooling**, where the new temperature $(T^{'})$ of the SA algorithm is computed using

$$T^{'} = \alpha T \ , \tag{3.6}$$

where α $(0 < \alpha < 1)$ denotes the cooling factor. Typically the value of α is chosen in the range 0.90 to 0.99. This cooling schedule has the advantage of being well understood, having a solid theoretical foundation, and being the most widely used annealing schedule. Our results obtained from using this standard cooling schedule will be used as a baseline for comparison with those using the other two schedules, which allow the rate of cooling to be varied.

The second annealing schedule we used is the method of **reheating as a function of cost** (RFC), which was used for timetabling by Abramson *et al.* [?], but the ideas behind it are due to Kirkpatrick *et al.* [?, ?] and White [?]. Before introducing this schedule we first summarize a few relevant points on the concept of specific heat (C_H). Specific heat is a measure of the variance of the cost (or energy) values of states at a given temperature. The higher the variance, the longer it presumably takes to reach equilibrium, and so the longer one should spend at the temperature, or alternatively, the slower one should lower the temperature.

Generally, in combinatorial optimization problems, phase transitions [?, ?] can be observed as sub-parts of the problem are resolved. In some of the work dealing with the traveling salesman problem using annealing [?], the authors often observe that the resolution of the overall structure of the solution occurs at high temperatures, and at low temperatures the fine details of the solution are resolved. As reported in [?], applying a reheating type procedure, depending on the phase, would allow the algorithm to spend more time in the low temperature phases, thus reducing the total amount of time required to solve a given problem.

In order to calculate the temperature at which a phase transition occurs, it is necessary to compute the specific heat of the system. A phase transition occurs at a temperature $T(C_H^{max})$ when the specific heat is maximal (C_H^{max}), and this triggers the change in the state ordering. If the best solution found to date has a high energy or cost then the super-structure may require re-arrangement. This can be done by raising the temperature to a level which is higher than the phase transition temperature $T(C_H^{max})$. Generally, the higher the current best cost, the higher the temperature which is required to escape the local minimum. To compute the aforementioned maximum specific heat, we employ the following steps [?, ?, ?].

At each temperature T, the annealing algorithm generates a set of configurations $\mathcal{C}(T)$. Let C_i denote the cost of configuration i, $C(T)$ is the average cost at temperature T, and $\sigma(T)$ is the standard deviation of the cost at T.

At temperature T, the probability distribution for configurations is:

$$P_i(T) = \frac{e^{\frac{-C_i}{kT}}}{\sum_j e^{\frac{-C_j}{kT}}}\,. \tag{3.7}$$

The average cost is computed as:

$$< C(T) >= \sum_{i \in \mathcal{C}} C_i P_i(T)\,. \tag{3.8}$$

Therefore, the average square cost is:

$$< C^2(T) >= \sum_{i \in \mathcal{C}} C_i^2 P_i(T)\,. \tag{3.9}$$

The variance of the cost is:

$$\sigma^2(T) =< C^2(T) > - < C(T) >^2 \ . \tag{3.10}$$

Now, the specific heat is defined as:

$$C_H(T) = \frac{\sigma^2(T)}{T^2}\,. \tag{3.11}$$

The temperature $T(C_H^{max})$ at which the maximum specific heat occurs, or at which the system undergoes a phase transition, can thus be found.

Reheating sets the new temperature to be

$$T = K \cdot C_b + T(C_H^{max})\,, \tag{3.12}$$

where K is a tunable parameter and C_b is the current best cost. Reheating is done when the temperature drops below the phase transition (the point of maximum specific heat) and there has been no decrease in cost for a specified number of iterations, i.e. the system gets stuck in a local minimum. Reheating increases the temperature above the phase transition (see equation 4.11), in order to produce enough of a change in the configuration to allow it to explore other minima when the temperature is reduced again.

The third cooling schedule we have tried is **adaptive cooling**. In this case, a new temperature is computed based on the specific heat, i.e. the standard deviation of all costs obtained at the current T. The idea here is to keep the system close to equilibrium, by cooling slower close to the phase transition, where the specific heat is large. There are many different ways of implementing this idea, we have chosen the approach taken by Huang *et al.* [?], which was shown to yield an efficient cooling schedule. Let T_j denote the current temperature, at step j of the annealing schedule. After calculating $\sigma(T_j)$ from equation 4.9, the new temperature T_{j+1} is computed as follows:

$$T_{j+1} = T_j \cdot e^{-\frac{aT_j}{\sigma(T_j)}}, \tag{3.13}$$

where a is a tunable parameter. Following suggestions by Otten and van Ginneken [?] and Diekmann *et al.* [?], $\sigma(T_j)$ is smoothed out in order to avoid any dependencies of the temperature decrement on large changes in the standard deviation σ. We used the following standard method to provide a smoothed standard deviation $\bar{\sigma}$:

$$\bar{\sigma}(T_{j+1}) = (1-\omega)\sigma(T_{j+1}) + \omega\sigma(T_j)\frac{T_{j+1}}{T_j} \tag{3.14}$$

and set ω to 0.95. This smoothing function is used because it follows (from the form of the Boltzmann distribution, see [?, ?]) that it preserves the key relationship:

$$\frac{d}{dT}C(T) = \frac{\bar{\sigma}^2(T)}{T^2} = C_H \tag{3.15}$$

Note that reheating can be used in conjunction with any cooling schedule. We have used it with adaptive cooling.

3.9.3 The Choice of Moves

The performance of any application of simulated annealing is highly dependent on the method used to select a new trial configuration of the system for the Metropolis update. In order for the annealing algorithm to work well, it must be able to effectively sample the parameter space, which can only be done with efficient moves.

The simplest method for choosing a move is to swap the rooms or timeslots of two randomly selected classes. However this is extremely inefficient, since most of the time random swapping of classes will increase the overall cost, especially if we are already close to obtaining a valid solution (i.e. at low temperature), and will likely be rejected in the Metropolis procedure. This low acceptance of the moves means this simple method is very inefficient, since a lot of computation is required to compute the change in cost and do the Metropolis step, only to reject the move.

What is needed is a strategy for choosing moves that are more likely to be accepted. A simple example is in the choice of room. If we randomly choose a new room from the list of all rooms, it will most likely be rejected, since it may be too small for the class, or an auditorium when, for example, a laboratory is needed. One possibility is to create a subset of all the rooms which fulfill the hard constraints on the room for that particular class, such as the size and type of room. Now we just make a random selection for a room for that class only from this subset of feasible rooms, with an acceptance probability that is sure to be much higher. In addition, each class in our data set comes with a "type-of-space-needed" tag which is used along with other information to assign the class to the right room. This effectively separates the updates into independent sets based on room type, so for example, laboratories are scheduled separately from lectures. In our method we carry out the scheduling of lectures first, followed by scheduling of laboratories making sure that during the course of this process no lecture and its associated laboratory are scheduled in the same time period.

In effect, we have embedded a simple expert system into the annealing algorithm in order to improve the choice of moves, as well as using a more complex expert system as a preprocessor for the annealing step. When used to choose the moves for annealing, the main function of the rule-based system is to ensure that all the trial moves satisfy the hard constraints. Many of the rules dealing with the medium and soft constraints are softened or eliminated, since reducing the cost of these constraints is done using the Metropolis update in the annealing algorithm.

Another of the modifications to the rule-based system is that while the version used in the preprocessor is completely deterministic, the version used in choosing the moves for annealing selects at random from multiple possibilities that satisfy the rules equally well. This extra freedom in choosing new schedules, plus the extra degree of randomness inherent in the annealing update, helps prevent the system from getting trapped in a local minimum before it can reach a valid schedule, which is the problem with the standard deterministic rule-based system.

To improve further on the move strategy, we can take the subset of possible move choices that we have created for each class, and choose from them probabilistically rather than randomly. There may be certain kinds of moves that are more likely to be effective, so our move strategy is to select these moves with a higher probability. For example, swapping a higher level class (e.g. graduate) with a lower level class (e.g. a first or a second year type) generally has a higher acceptance, since there is little overlap between students taking these classes. Furthermore, we have experimented with two kinds of swaps, those that only involve classes offered by the same department or college and the second, swaps between classes of different departments and colleges.

Generally, the swap methods we have taken here can be considered as heuristics for pruning the neighborhood or narrowing the search space, which provides much more efficient moves and in turn an overall improvement in the results.

3.10 Experimental Results

Our computations were done with a number of goals in mind. The main objective was to provide a schedule which satisfied all hard constraints and minimized the cost of medium and soft constraints, using real-life data sets for a large university. We also aimed to find an acceptable set of annealing parameters and move strategies for general timetabling problems of this kind, and to study the effect of using a preprocessor to provide the annealing program with a good starting point. Finally, we wanted to make a comparison of the performance of the three different cooling schedules, geometric cooling, adaptive cooling, and reheating based on cost.

We spent quite some time finding optimal values for the various parameters for the annealing schedule, such as the initial temperature, the parameters controlling the rate of cooling (α for geometric cooling, a for adaptive cooling) and reheating (K), and the number of iterations at each temperature (for more details, see Ref. [?]). Johnson *et al.* [?] noted in their SA implementation for the traveling salesman problem (TSP) that the number of steps at each temperature (or the size of the Markov chain) needed to be at least proportional to the "neighborhood" size in order to maintain a high-quality result. From our experiments we found the same to be true for the scheduling problem, even though it is very different from the TSP. Furthermore, in a few tests for one semester we fixed the number of classes and professors but varied the number of rooms and time slots, and found that the final result improves as the number of iterations in the Markov chain becomes proportional to a combination of the number of classes, rooms and time slots. We also observed the same behavior when we fixed the number of rooms and time slots but varied number of classes.

Our study case involved real scheduling data covering three semesters at Syracuse University. The size and type of the three-semester data is shown in Table 4.1. Nine types of rooms were used: auditoriums, classrooms, computer clusters, conference rooms, seminar rooms, studios, laboratories, theaters, and unspecified types. Staff and teaching assistants are considered part of the set of professors. Third semester (summer) data was much smaller than other semesters, however, there were additional space and time constraints and fewer available rooms. Our data was quite large in comparison to data used by other researchers. For example, high school data used by Peterson and colleagues [?, ?] consists of approximately 1000 students, 20 different possible majors, and an overall periodic school schedule (over weeks). In the case of Abramson *et al.* [?], their data set was created randomly and was relatively small, and they stated that problems involving more than 300 tuples were very difficult to solve.

Table 4.1 lists all major components of the data we have used. Timetabling problems can be characterized by their *sparseness*. After the required number of lessons N_l have been scheduled, there will be $N_{sp} = (N_x N_t - N_l)$ spare space-time slots, hence, the sparseness ratio of the problem is defined as the ratio $N_{sp}/(N_x N_t)$. The denser the problem, the lower the sparseness ratio, and the harder the problem is to solve. Also, for dense problems, there is an additional correlation involving the problem size. Table 4.2 shows the sparseness of the three-semester data. For university scheduling, the sparseness ratio generally decreases as the data size (particularly the number of classes) increases, so the problem becomes harder to solve. Including student preferences makes the problem much harder, but these are viewed as medium constraints and thus are not necessarily satisfied in a valid solution.

Our overall results are shown in Tables 4.3 and 4.4. These tables show the percentage of classes that could be scheduled in accordance with the hard constraints. In each case (apart from the expert system, which is purely deterministic), we have done 10 runs (with the same parameters, just different random numbers), and the tables show the average of the 10 runs, as well as the best and worst results. The MFA results are different only due to having different initial conditions. Each simulated annealing run takes about 10 to 20 hours on a Unix workstation, while a single MFA run takes approximately an hour and an expert system run takes close to two hours.

As expected, each of the methods did much better for the third (summer) semester data, which has a higher sparseness ratio. Our results also confirm what we expected for the different cooling schedules for simulated annealing, in that adaptive cooling performs better than geometric cooling, and reheating improves the result even further.

When a random initial configuration is used, simulated annealing performs very poorly, even worse than the expert system (ES). However, there is a dramatic improvement in performance when a preprocessor is used to provide a good starting point for the annealing. In that case, using the best cooling schedule of adaptive cooling with reheating as a function of cost, we are able to find a valid class schedule every time.

In the case of mean-field annealing, the overall results are generally below those of SA and ES. In addition, we have found in the implementation of this method that the results were quite sensitive to the size of the data as well the type of constraints involved. If we confine ourselves to the set of hard constraints, the results are as good as or even better than the other methods. However if we take into account the medium and soft constraints, that is, the overall cost function, this method does not perform as well.

Student preferences are included only as medium constraints in our implementation, meaning that these do not have to be satisfied for a valid solution, but they have a high priority. For the valid schedules we have produced, approximately 75% of the student preferences were satisfied. This is reasonably good (particularly since other approaches do not deal with student preferences at all), but we are working to improve upon this result.

3.11 Conclusions

We have successfully applied simulated annealing to the difficult problem of academic scheduling for a large university. Feasible schedules were obtained for real data sets, including student preferences, without requiring enormous computational effort.

Mean-field annealing works well for small scheduling problems, but does not appear to scale well to large problems with many complex constraints. For this problem, both simulated annealing and the rule-based system were more effective than MFA. It is more difficult to tune the parameters for MFA than for simulated annealing, and because of the complexity and size of the Potts neural encoding, there seems to be no clear way of preserving the state of a good initial configuration provided by a preprocessor when using MFA.

Using a preprocessor to provide a good initial state greatly improved the quality of the results for simulated annealing. In theory, using a good initial state should not be necessary, and any initial state should give a good result, however in practice, we do not have an ideal cooling schedule for annealing, or an ideal method for choosing trial moves and efficiently exploring the search space, and there are restrictions on how long the simulation can take. In general, for very hard problems with large parameter spaces that can be difficult to search efficiently, and for which very slow cooling would be much too time-consuming, we might expect that a good initial solution would be helpful. We used a fairly complex rule-based expert system for the preprocessor, however the type of preprocessor may not be crucial. Other fast heuristics could possibly be used, for example a graph coloring approach [?], or it may be possible to just utilize the schedule from the same semester for the previous year. A modified version of the rule-based system was used to choose the trial moves for the simulated annealing, and the high acceptance rate provided by this system was crucial to obtaining good results.

As expected, for the simulated annealing, adaptive cooling performed better than geometric cooling, and using reheating improved the results even further. The best results were obtained using simulated annealing with adaptive cooling and reheating as a function of cost, and with a rule-based preprocessor to provide a good initial solution. Using this method, and with careful selection of parameters and update steps, we were able to generate solutions to the class scheduling problem using real data for a large university. None of the other methods were able to provide a complete solution.

Our main conclusion from this work is that simulated annealing, with a good cooling schedule, optimized parameters, carefully selected update moves, and a good initial solution provided by a preprocessor, can be used to solve the academic scheduling problem at a large university, including student preferences. Similar approaches should prove fruitful for other difficult scheduling problems.

Table 3.1: Size of the data set for each of the three semesters.

	First Semester	Second Semester	Third Semester
Rooms	509	509	120
Classes	3839	3590	687
Professors	1190	1200	334
Students	13653	13653	2600
Buildings	43	43	11
Schools and/or Colleges	20	21	17
Departments or Course Prefixes	143	141	108
Areas of Study (majors)	200	200	200

Table 3.2: The sparseness ratios of the problem for the data sets for each of the three semesters. Lower values indicate a harder problem.

Academic Time Period	Sparseness ratio
First Semester	0.50
Second Semester	0.53
Third Semester	0.62

Table 3.3: Percentage of classes scheduled using the different methods. The averages and highest and lowest values were obtained using 10 independent runs for simulated annealing (SA) and mean-field annealing (MFA). The expert system (ES) is deterministic so the results are from a single run. *No preprocessor was used with the three methods.*

Academic Time Period	Algorithm	Scheduled (average) %	Highest Scheduled %	Lowest Scheduled %
First Semester	SA (geometric)	65.00	67.50	56.80
	SA (adaptive)	67.80	70.15	61.20
	SA (cost-based)	70.20	72.28	68.80
	ES	76.65	76.65	76.65
	MFA	65.60	71.00	61.00
Second Semester	SA (geometric)	65.65	68.00	57.10
	SA (adaptive)	68.50	70.10	60.77
	SA (cost-based)	75.14	77.68	70.82
	ES	79.00	79.00	79.00
	MFA	67.20	75.00	65.00
Third Semester	SA (geometric)	83.10	86.44	68.50
	SA (adaptive)	85.80	89.00	70.75
	SA (cost-based)	91.20	95.18	85.00
	ES	96.80	96.80	96.80
	MFA	88.00	95.00	82.00

Table 3.4: Percentage of scheduled classes, averaged over 10 runs of the same initial temperature and other parameters, for three terms using simulated annealing *with an expert system as preprocessor.*

Academic Time Period	Algorithm	Scheduled (average) %	Highest Scheduled %	Lowest Scheduled %
First Semester	SA (geometric)	93.90	95.12	85.20
	SA (adaptive)	98.80	99.20	95.00
	SA (cost-based)	100.0	100.0	100.0
Second Semester	SA (geometric)	95.00	98.95	89.40
	SA (adaptive)	99.00	99.50	98.50
	SA (cost-based)	100.0	100.0	100.0
Third Semester	SA (geometric)	97.60	98.88	90.90
	SA (adaptive)	100.0	100.0	100.0
	SA (cost-based)	100.0	100.0	100.0

Chapter 4

A Comparison of Annealing Techniques for Academic Course Scheduling

4.1 Summary

In this study we have tackled the NP-hard problem of academic class scheduling (or timetabling) at the university level. We have investigated a variety of approaches based on simulated annealing, including mean-field annealing, simulated annealing with three different cooling schedules, and the use of a rule-based preprocessor to provide a good initial solution for annealing. The best results were obtained using simulated annealing with adaptive cooling and reheating as a function of cost, and a rule-based preprocessor. This approach enabled us to obtain valid schedules for the timetabling problem for a large university, using a complex cost function that includes student preferences. None of the other methods were able to provide a complete valid schedule.

4.2 Introduction

The primary objective of this study is to derive an approximate solution to the problem of university class scheduling, or timetabling, which can be summarized as follows: given data sets of classes and their days, enrollments, and instructors; rooms and their capacities, types, and locations; distances between buildings; priorities of each building for different departments; and students and their class preferences; the problem is to construct a feasible class schedule satisfying all the hard constraints and minimizing the medium and soft constraints. Hard constraints are space and time constraints that must be satisfied, such as scheduling only one class at a time for any teacher, student, or classroom. Medium and soft constraints are student and teacher preferences that should be satisfied if possible.

The timetabling problem (TTP) is a high-dimensional, non-Euclidean, multi-constraint combinatorial optimization problem, and is consequently very difficult to solve. This problem has been tackled by many researchers, mostly in the field of operations research. A number of different heuristics have been tried on different instances of the problem, from high school to university course scheduling (see the reviews by de Werra [?] and Shaerf [?] and the papers collected in Ref. [?]). For small to medium size problems, such as exam scheduling, high school scheduling, or course scheduling for a university department, many of these methods work well. However no particular method has yet been shown to produce good results for real-world problems on a much larger scale, such as scheduling all courses for a large university, which we address in this paper. Also, we are not aware of any large scale study that takes into account constraints due to student preferences, as we have done.

We have used data for classes at Syracuse University. Currently this problem is handled by the university scheduling department in a semi-automated fashion. A scheduling program is used to find a partial solution, and substantial manual effort is required to iterate towards a final solution. Also, when scheduling a certain semester (e.g. fall 1996), a template of a previous semester (e.g. fall 1995) is used as part of the input data.

We have applied the following optimization techniques to this problem:

1. A rule-based expert system.
2. Mean-field annealing.
3. Simulated annealing with geometric cooling.
4. Simulated annealing with adaptive cooling.
5. Simulated annealing with adaptive cooling and reheating as a function of cost.
6. Simulated annealing (using each of the three different cooling schedules) with a rule-based preprocessor to provide a good initial solution.

The best results were obtained using simulated annealing with adaptive cooling and reheating as a function of cost, and with a rule-based preprocessor to provide a good initial solution. Using this method, and with careful selection of parameters and update moves, we were able to generate solutions to the class scheduling problem using real data for a large university. None of the other methods were able to provide a complete valid solution.

4.3 The Timetabling Problem

Timetabling is the assignment of time slots to a set of events, subject to constraints on these assignments. The NP-complete *professors and classes* timetabling problem [**?**, **?**, **?**] is a constraint satisfaction problem that can be briefly stated as follows:

For a certain school with N_p professors, N_q classes, N_x classrooms and lecture halls, and N_s students, it is required to schedule N_l professor-class pairs within a time limit of N_t time slots producing a legal schedule. A legal schedule needs to be found such that no professor, class, or student is in more than one place at a time, and no room is expected to accommodate more than one lesson at a time or more students than its capacity.

The constraints for this problem can be hard, medium or soft. The medium and soft constraints have an associated cost (or penalty), and if they are not satisfied, the goal is to minimize this cost. Soft constraints have a lower priority (and thus lower cost) than medium constraints. The hard constraints must be satisfied, so their associated cost must be reduced to zero. A feasible schedule is one that satisfies all the hard constraints.

Hard constraints are usually constraints that physically cannot be violated. This includes events that must not overlap in time, such as:

- classes taught by the same professor,
- classes held in the same room,
- a class and a recitation or a lab of the same class.

Another examples are space or room constraints:

- A class cannot be assigned to a particular room unless the capacity of the room is greater than or equal to the class enrollment.
- Some classes, such as laboratories, require a certain type of room.

Medium constraints are usually considered to be those constraints that fall into the gray area between the hard and soft constraints [?]. In our implementation, we define medium constraints to be constraints such as time and space conflicts which, like hard constraints, cannot physically be violated (for example, it is not possible for one person to be in two different classes at the same time). However we consider these constraints to be medium rather than hard if they can be avoided by making adjustments to the specification of the problem. The primary example is student preferences. We cannot expect to be able to satisfy all student class preferences, in some cases, certain students will have to adjust their preferences since certain classes will clash, or will be oversubscribed.

Medium constraints have a high penalty attached to them, although not as high as that associated with the hard constraints. In the final schedule the penalty of these constraints should be minimized and preferably reduced to zero. Some examples of medium constraints are:

- Avoid time conflicts for classes with students in common.
- Eligibility criteria for the class must be met.
- Do not enroll athletes in classes that conflict with their sport practice time (of course, depending on the sport).

Soft constraints are preferences that do not deal with time conflicts, and have a lower penalty (or cost) associated with them. We aim to minimize the cost, but do not expect to be able to reduce it to zero. Some examples are:

- For each student, balance the three-day (*Mon, Wed, Fri*) as well as the two-day (*Tue, Thu*) schedules.
- Balance or spread out the lectures over the week.
- Classes may request contiguous time slots.
- Balance enrollment in multi-section classes.
- Lunch and other break times may be specified.
- Professors may request periods in which their classes are not taught.
- Professors may have preferences for specific rooms or types of rooms.
- Minimize the distance between the room where the class is assigned and the building housing its home department.

Some soft constraints may have higher priority (and thus higher cost) than others. For example, preferences involving teachers will have higher priority than the preferences of students.

The **cost function** measures the quality of the current schedule and generally involves the weighted sum of penalties associated with different types of constraint violations. The aim of the optimization technique is to minimize the cost function.

4.4 Mean-Field Annealing

One of the potential drawbacks of using simulated annealing for hard optimization problems is that finding a good solution can often take an unacceptably long time. Mean-field annealing (MFA) attempts to avoid this problem by using a deterministic approximation to simulated annealing, by attempting to average over the statistics of the annealing process. The result is improved execution speed at the expense of solution quality. Although not strictly a continuous descent technique, MFA is closely related to the Hopfield neural network [?, ?].

Mean-field annealing has been successfully applied to high school class scheduling [?]. For scheduling, it is advantageous to use a Potts neural encoding to specify discrete neural variables (or neurons) for the problem. This is defined in its simplest form as a mapping of events onto space-time slots, for example an event i, in this case a professor-class pair (p, q), is mapped onto a space-time slot a, in this case a classroom-timeslot pair (x, t). Now, the Potts neurons S_{ia} are defined to be 1 if event i takes place in space-time slot a, and 0 otherwise. In this way, the constraints involved can be embedded in the neural net in terms of the weights $w_{i,j}$ of the neural network, which encode a Potts normalization condition such as $\sum_a S_{ia} = 1$.

For a full derivation of the mean-field annealing algorithm from its roots in statistical physics, see Hertz *et al.* [?] or Peterson *et al.* [?]. Here we will just give a brief overview of the method. The basic idea is that it is possible to approximate the actual cost or energy function E, which is a function of discrete neural variables S_{ia}, by an effective energy function $E^{'}$ that can be represented in terms of continuous variables U_{ia} and V_{ia}. These are known as mean field variables, since V_{ia} is an approximation to the average value of S_{ia} at a given temperature T.

This approach effectively smooths out the energy function and makes it easier to find the minimum value, which is obtained by solving the saddle point equations $\frac{\partial E^{'}}{\partial V_{ia}} = 0$ and $\frac{\partial E^{'}}{\partial U_{ia}} = 0$, which generate a set of self-consistent mean field theory (MFT) equations in terms of the mean field variables U and V:

$$U_{ia} = -\frac{1}{T}\frac{\partial E}{\partial V_{ia}} \tag{4.1}$$

$$V_{ia} = \frac{e^{U_{ia}}}{\sum_b e^{U_{ib}}}. \tag{4.2}$$

The MFA algorithm involves solving equations 4.1 and 4.2 at a series of progressively lower temperatures T: this process is known as temperature annealing. The critical temperature T_c, which sets the scale of T, is estimated by expanding equation 4.2 around the trivial fixed-point [?, ?] $V_{ia}^{(0)} = \frac{1}{N_a}$, where N_a is the number of possible states of each of the network neurons. For example, for the events defined by professor-class pairs (p, q) mapped onto classroom-timeslots (x, t), we have $N_p N_q$ neurons, each of which has $N_x N_t$ possible states, in which case $V_{pq;xt}^{(0)} = \frac{1}{N_x N_t}$.

Equations 4.1 and 4.2 can be solved iteratively using either synchronous or serial updating. The iterative dynamics to evolve the mean field variables toward a self-consistent solution is explained in detail by Peterson *et al.* [?]. The solutions correspond to stable states of the Hopfield network [?]. Observe from equation 4.2 that any solution to the MF equations respects a continuous version of the Potts condition

$$\sum_a V_{ia} = 1 \quad \forall\, i. \tag{4.3}$$

4.4.1 The Mean-Field Annealing Algorithm

The generic MFA algorithm appears in Figure 4.1. At high temperatures T, the mean-field solutions will be states near the fixed-point symmetrical maximum entropy state $V_{ia} = 1/N_a$. At low temperatures, finding a mean-field solution will be equivalent to using the Hopfield model, which is highly sensitive to the initial conditions and known to be ineffective for hard problems [?]. MFA improves over the Hopfield model by using annealing to slowly decrease the temperature in order to sidestep these problems.

These characteristics are similar to those of simulated annealing, which is no surprise since both it and the mean-field method compute thermal averages over

Gibbs distributions of discrete states, the former stochastically and the latter through a deterministic approximation. It is therefore natural to couple the mean-field method with the concept of annealing from high to low temperatures.

In addition to the structure of the energy function, there are three major interdependent issues which arise in completely specifying a mean-field annealing algorithm for a timetabling problem:

- The values of the coefficients of terms in the energy function.
- The types of dynamics used to find solutions of the MFT equations at each T.
- The annealing schedule details, i.e. the initial temperature $T(0)$, the rules for deciding when to reduce T and by how much, and the termination criteria.

Peterson *et al.* [?] introduced a quantity called *saturation*, Σ, defined as

$$\Sigma = \frac{1}{N_i} \sum_{ia} V_{ia}^2 \quad , \tag{4.4}$$

where N_i is the number of events (in this case the number of professor-class pairs). This characterizes the degree of clustering of events in time and/or space, $\Sigma_{min} = \frac{1}{N_a}$ corresponds to high temperature, whereas $\Sigma_{max} = 1$ means that all the V_{ia} have converged to 0 or 1 values, indicating that each event has been assigned to a space-time slot.

The first step of Figure 4.1 is to map the constraints of the problem into the neural net connection weights. In our implementation, at each T_n the MFA algorithm (Figure 4.1) performs one update per neural variable (defined as one sweep) with sequential updating using equations 4.1 and 4.2. After reaching a saturation value close to 1 (we chose $\Sigma = 0.99$) we check whether the obtained solutions are valid, i.e. $E_{hard} = 0$. If this is not the case the network is re-initialized and is allowed to resettle. We repeat this procedure a number of times until the best solution is found. A similar procedure was carried out on high school scheduling by Peterson *et al.* [?].

The MFA implementation was a little more complicated than the implementation of simulated annealing and the expert system, since it had many more parameters to

1. Choose a problem and encode the constraints into weights $\{w_{ij}\}$.
2. Find the approximate phase transition temperature by linearizing equation (4.2).
3. Add a self-coupling β-term if necessary. In a neural net, this corresponds to a feedback connection from a neuron to itself.
4. Initialize the neurons V_{ia} to high temperature values $\frac{1}{N_a}$ plus a small random term such as $rand[-1, 1] \times 0.001$; and set $T(0) = T_c$.
5. Until $(\Sigma \geq 0.99)$ do:
   - At each $T(n)$, update all U_{ia} and V_{ia} by iterating to a solution of the mean field equations.
   - $T(n+1) = \alpha T(n)$, we chose $\alpha = 0.9$
6. The discrete values S_{ia} that specify the schedule are obtained by rounding the mean field values V_{ia} to the nearest integer (0 or 1).
7. Perform *greedy heuristics* if needed to account for possible imbalances or rule violations.

Figure 4.1: The Generic Mean-Field Annealing Algorithm

handle, and it was often more difficult to find optimal values for these parameters. For example, one complication is the computation of the critical temperature T_c, which involved an iterative procedure of a linearized dynamic system. On the other hand, we observed that the convergence time was indeed much less than any of the convergence times of the simulated annealing using the three annealing schedules studied. For more details on our MFA implementation, see Ref. [?].

4.5 The Rule-Based System

We have implemented a fairly complex rule-based expert system for solving the timetabling problem, for three reasons. Firstly, it gives us a benchmark as to how well other methods do in comparison to this standard technique. Secondly, a simplified version of the rule-based system is used to provide sensible choices for moves in the simulated annealing algorithm, rather than choosing swaps completely at random, and this greatly improves the proportion of moves that are accepted. Thirdly, we have used this system as a preprocessor for simulated annealing, in order to provide a good initial solution.

Simulated annealing is a very time-consuming, computationally intensive procedure. Using an expert system as a preprocessor is a way of quickly providing a good starting point for the annealing algorithm, which reduces the time taken in the annealing procedure, and improves the quality of the result. Our results clearly support this rationale for the case of academic scheduling.

The rule-based expert system consists of a number of rules (or heuristics) and conventional recursion to assist in carrying out class assignments. We have developed this system specifically for the problem of academic scheduling. The basic data structures or components of the system are:

1. Distance matrix of values between each academic department and every other building under use for scheduling.

2. Class data structure of each class scheduled anywhere in campus. These structures are capable of linking with each other.

3. Room data structure of each room (regardless of type) involved in the scheduling process. Like classes, room structures are also linked with each other.

4. Data structures for time periods to keep track of which hour or time slot was occupied and which was not.

5. Department inclusion data structure giving department inclusion within other larger departments or colleges.

6. Students structures indicating classes of various degree of requirements and preferences for each student.

The basic function of the system is as follows: given data files of classes, rooms and buildings, department-to-building distance matrix, students data, and the inclusion data, using the abovementioned data structures, the system builds an internal database which in turn is used in carrying out the scheduling process. This process involves a number of essential sub-processes such as checking the distances between buildings, checking building, room type and hours occupied, checking and comparing time slots for any conflicts, checking rooms for any space conflict, and keeping track of and updating the hours already scheduled.

The rule-based system uses an iterative approach. The basic procedure for each iteration is as follows. The scheduling of classes is done by department, so each iteration consists of a loop over all departments. The departments are chosen in order of size, with those having the most classes being scheduled first. The system first loops over all the currently unscheduled classes, and attempts to assign them to the first unoccupied room and timeslot that satisfies all the rules governing the constraints. Since constraints involving capacity of rooms are very difficult to satisfy, larger classes are scheduled first, to try to avoid not having large enough rooms later for those class sections with large enrollments.

In some cases the only rooms and timeslots that satisfy all the rules will already be occupied by previously scheduled classes. In that case, the system attempts to move one of these classes into a free room and timeslot, to allow the unscheduled class to be scheduled.

Next, the system searches through all the scheduled classes, and selects those that have a high cost, by checking the medium and soft constraints such as how closely the room size matches the class size, how many students have time conflicts, whether the class is in a preferred time period or a preferred building, and so on. Selecting threshold values for defining what is considered a "high" cost in each case is a subjective procedure, but it is straightforward to choose reasonable values. When a poorly scheduled class is identified, the system searches for a class to swap it with, so that the hard constraints are still satisfied, but the overall cost of the medium and soft constraints is reduced.

This process of swapping rooms continues provided all the rules are satisfied and no "cycling" (swapping of the same classes) occurs. Once all the departments have been considered, this completes one iteration. The system continues to follow this iterative procedure until a complete iteration produces no changes to the schedule.

There are many rules dealing with space and hours, type of room, and priority of room. Many are quite complex, but some of the basic rules, such as those implementing the hard constraints, can be quite straightforward – for example, the following is the basic rule for dealing with time and space conflicts for a room:

IF [room(capacity) > class(space-requested)] and [no time conflict in this room] THEN assign the room to the class.

When the rule-based system is used as a preprocessor, it produces a partial schedule as an output, since it is usually unable to assign all of the given classes to rooms and times slots. The output is divided into two parts: the first consists of classes, with their associated professors and students, assigned to various rooms; and the second is a list of classes that could not be assigned due to constraint conflicts.

4.6 Simulated Annealing

Simulated annealing (SA) has been widely used for tackling different combinatorial optimization problems, particularly academic scheduling [?, ?, ?]. The basic algorithm is described in Figure 4.2. The results obtained depend heavily on the cooling schedule used. We initially used the most commonly known and used schedule, which is the geometric cooling, but later tried adaptive cooling, as well as the method of geometric reheating based on cost [?].

A comprehensive discussion of the theoretical and practical details of SA is given in [?, ?, ?, ?]. It suffices here to say that the elementary operation in the Metropolis method for a combinatorial problem such as scheduling is the generation of some new candidate configuration, which is then automatically accepted if it lowers the cost (C), or accepted with probability $\exp(-\Delta C/T)$, where T is the temperature, if it would increase the cost by $\Delta(C)$. Also, in Figure 4.2, s is the current schedule and

1. Generate an initial schedule s.
2. Set the initial best schedule $s^* = s$.
3. Compute cost of $s : C(s)$.
4. Compute initial temperature T_0.
5. Set the temperature $T = T_0$.
6. While *stop criterion* is not satisfied do:
   (a) Repeat *Markov chain length* (M) times:
      i. Select a random neighbor s' to the current schedule, $(s' \subset \mathcal{N}_s)$.
      ii. Set $\Delta(C) = C(s') - C(s)$.
      iii. If $(\Delta(C) \leq 0$ {downhill move}):
         - Set $s = s'$.
         - If $C(s) < C(s^*)$ then set $s^* = s$.
      iv. If $(\Delta(C) > 0$ {uphill move}):
         - Choose a random number r uniformly from $[0, 1]$.
         - If $r < e^{-\Delta(C)/T}$ then set $s = s'$.
   (b) Reduce (or update) temperature T.
7. Return the schedule s^*.

Figure 4.2: The Simulated Annealing Algorithm

s' is a neighboring schedule obtained from the current neighborhood space $(\mathcal{N}_s)$ by swapping two classes in time and/or space.

Thus the technique is essentially a generalization of the local optimization strategy, where, at non-zero temperatures, thermal excitations can facilitate escape from local minima.

The SA algorithm has advantages and disadvantages compared to other global optimization techniques. Among its advantages are the relative ease of implementation, the applicability to almost any combinatorial optimization problem, the ability to provide reasonably good solutions for most problems (depending on the cooling schedule and update moves used), and the ease with which it can be combined with other heuristics, such as expert systems, forming quite useful hybrid methods for tackling a range of complex problems. SA is a robust technique, however, it does have some drawbacks. To obtain good results the update moves and the various tunable parameters used (such as the cooling rate) need to be carefully chosen, the runs often require a great deal of computer time, and many runs may be required.

Depending on the problem to which it is applied, SA appears competitive with many of the best heuristics, as shown in the work of Johnson *et al.* [?].

4.6.1 Timetabling Using the Annealing Algorithm

The most obvious mapping of the timetabling problem into the SA algorithm involves the following constructs:

1. a **state** is a timetable containing the following sets:
   - P: a set of professors.
   - C: a set of classes.
   - S: a set of students.
   - R: a set of classrooms.
   - I: a set of time intervals.
2. a **cost** or "energy" $\mathrm{E}(P, C, S, R, I)$ such that:
   - $\mathrm{E}(P)$: is the cost of assigning more than maximum number of allowed classes M_p to the same professor, plus scheduling one or more classes that cause a conflict in the professor's schedule.
   - $\mathrm{E}(C)$: is the cost of scheduling certain classes at/within the same time period in violation of the exclusion constraint, for example.

- E(S): is the cost of having two or more classes conflict in time; plus cost of having in the schedule one or more classes that really don't meet the student's major, class requested, or class requirements; plus the cost of not having the classes evenly spread out over the week, etc.
- E(R): is the cost resulting from assigning room(s) of the wrong size and/or type to a certain class.
- E(I): is the cost of having more or less time periods than required, plus cost of an imbalanced class assignments (a certain period will have more classes assigned to than others, etc.).

3. A **swap** (or a move) is the exchange of one or more of the following: class c_i with class c_j in the set C with respect to time periods I_i and I_j, and/or with respect to classroom R_i and R_j, respectively. Generally, this step is referred to as class swapping.

Along with all of the necessary constraints, the simulated annealing algorithm also takes as input data the following: the preprocessor output in the form of lists of scheduled and non-scheduled classes and their associated professors and room types, a list of rooms provided by the registrar's office, a department to building distance matrix, a list of students and their class preferences, and a list of classes that are not allowed to be scheduled simultaneously.

To use simulated annealing effectively, it is crucial to use a good cooling schedule, and a good method for choosing new trial schedules, in order to efficiently sample the search space. We have experimented with both these areas, which are discussed in the following sections.

4.6.2 The Annealing Schedules

Three annealing schedules have been used in our experiments to update the temperature of the SA algorithm in Figure 4.2: geometric cooling, adaptive cooling, and adaptive reheating as a function of cost.

The first schedule we have used is **geometric cooling**, where the new temperature ($T^{'}$) of the SA algorithm is computed using

$$T^{'} = \alpha T \ , \tag{4.5}$$

where α ($0 < \alpha < 1$) denotes the cooling factor. Typically the value of α is chosen in the range 0.90 to 0.99. This cooling schedule has the advantage of being well understood, having a solid theoretical foundation, and being the most widely used annealing schedule. Our results obtained from using this standard cooling schedule will be used as a baseline for comparison with those using the other two schedules, which allow the rate of cooling to be varied.

The second annealing schedule we used is the method of **reheating as a function of cost** (RFC), which was used for timetabling by Abramson *et al.* [?], but the ideas behind it are due to Kirkpatrick *et al.* [?, ?] and White [?]. Before introducing this schedule we first summarize a few relevant points on the concept of specific heat (C_H). Specific heat is a measure of the variance of the cost (or energy) values of states at a given temperature. The higher the variance, the longer it presumably takes to reach equilibrium, and so the longer one should spend at the temperature, or alternatively, the slower one should lower the temperature.

Generally, in combinatorial optimization problems, phase transitions [?, ?] can be observed as sub-parts of the problem are resolved. In some of the work dealing with the traveling salesman problem using annealing [?], the authors often observe that the resolution of the overall structure of the solution occurs at high temperatures, and at low temperatures the fine details of the solution are resolved. As reported in [?], applying a reheating type procedure, depending on the phase, would allow the algorithm to spend more time in the low temperature phases, thus reducing the total amount of time required to solve a given problem.

In order to calculate the temperature at which a phase transition occurs, it is necessary to compute the specific heat of the system. A phase transition occurs at a temperature $T(C_H^{max})$ when the specific heat is maximal (C_H^{max}), and this triggers the change in the state ordering. If the best solution found to date has a high energy or cost then the super-structure may require re-arrangement. This can be done by raising the temperature to a level which is higher than the phase transition temperature $T(C_H^{max})$. Generally, the higher the current best cost, the higher the temperature which is required to escape the local minimum. To compute the aforementioned maximum specific heat, we employ the following steps [?, ?, ?].

At each temperature T, the annealing algorithm generates a set of configurations $\mathcal{C}(T)$. Let C_i denote the cost of configuration i, $C(T)$ is the average cost at temperature T, and $\sigma(T)$ is the standard deviation of the cost at T.

At temperature T, the probability distribution for configurations is:

$$P_i(T) = \frac{e^{\frac{-C_i}{kT}}}{\sum_j e^{\frac{-C_j}{kT}}}. \tag{4.6}$$

The average cost is computed as:

$$< C(T) >= \sum_{i \in \mathcal{C}} C_i P_i(T). \tag{4.7}$$

Therefore, the average square cost is:

$$< C^2(T) >= \sum_{i \in \mathcal{C}} C_i^2 P_i(T). \tag{4.8}$$

The variance of the cost is:

$$\sigma^2(T) =< C^2(T) > - < C(T) >^2 \;. \tag{4.9}$$

Now, the specific heat is defined as:

$$C_H(T) = \frac{\sigma^2(T)}{T^2}. \tag{4.10}$$

The temperature $T(C_H^{max})$ at which the maximum specific heat occurs, or at which the system undergoes a phase transition, can thus be found.

Reheating sets the new temperature to be

$$T = K \cdot C_b + T(C_H^{max}), \tag{4.11}$$

where K is a tunable parameter and C_b is the current best cost. Reheating is done when the temperature drops below the phase transition (the point of maximum specific heat) and there has been no decrease in cost for a specified number of iterations, i.e. the system gets stuck in a local minimum. Reheating increases the temperature above the phase transition (see equation 4.11), in order to produce enough of a change in the configuration to allow it to explore other minima when the temperature is reduced again.

The third cooling schedule we have tried is **adaptive cooling**. In this case, a new temperature is computed based on the specific heat, i.e. the standard deviation of all costs obtained at the current T. The idea here is to keep the system close to equilibrium, by cooling slower close to the phase transition, where the specific heat is large. There are many different ways of implementing this idea, we have chosen the approach taken by Huang *et al.* [**?**], which was shown to yield an efficient cooling schedule. Let T_j denote the current temperature, at step j of the annealing schedule. After calculating $\sigma(T_j)$ from equation 4.9, the new temperature T_{j+1} is computed as follows:

$$T_{j+1} = T_j \cdot e^{-\frac{aT_j}{\sigma(T_j)}} , \tag{4.12}$$

where a is a tunable parameter. Following suggestions by Otten and van Ginneken [**?**] and Diekmann *et al.* [**?**], $\sigma(T_j)$ is smoothed out in order to avoid any dependencies of the temperature decrement on large changes in the standard deviation σ. We used the following standard method to provide a smoothed standard deviation $\bar{\sigma}$:

$$\bar{\sigma}(T_{j+1}) = (1-\omega)\sigma(T_{j+1}) + \omega\sigma(T_j)\frac{T_{j+1}}{T_j} \tag{4.13}$$

and set ω to 0.95. This smoothing function is used because it follows (from the form of the Boltzmann distribution, see [**?**, **?**]) that it preserves the key relationship:

$$\frac{d}{dT}C(T) = \frac{\bar{\sigma}^2(T)}{T^2} = C_H \tag{4.14}$$

Note that reheating can be used in conjunction with any cooling schedule. We have used it with adaptive cooling.

4.6.3 The Choice of Moves

The performance of any application of simulated annealing is highly dependent on the method used to select a new trial configuration of the system for the Metropolis update. In order for the annealing algorithm to work well, it must be able to effectively sample the parameter space, which can only be done with efficient moves.

The simplest method for choosing a move is to swap the rooms or timeslots of two randomly selected classes. However this is extremely inefficient, since most of the time random swapping of classes will increase the overall cost, especially if we are already close to obtaining a valid solution (i.e. at low temperature), and will likely be rejected in the Metropolis procedure. This low acceptance of the moves means this simple method is very inefficient, since a lot of computation is required to compute the change in cost and do the Metropolis step, only to reject the move.

What is needed is a strategy for choosing moves that are more likely to be accepted. A simple example is in the choice of room. If we randomly choose a new room from the list of all rooms, it will most likely be rejected, since it may be too small for the class, or an auditorium when, for example, a laboratory is needed. One possibility is to create a subset of all the rooms which fulfill the hard constraints on the room for that particular class, such as the size and type of room. Now we just make a random selection for a room for that class only from this subset of feasible rooms, with an acceptance probability that is sure to be much higher. In addition, each class in our data set comes with a "type-of-space-needed" tag which is used along with other information to assign the class to the right room. This effectively separates the updates into independent sets based on room type, so for example, laboratories are scheduled separately from lectures. In our method we carry out the scheduling of lectures first, followed by scheduling of laboratories making sure that during the course of this process no lecture and its associated laboratory are scheduled in the same time period.

In effect, we have embedded a simple expert system into the annealing algorithm in order to improve the choice of moves, as well as using a more complex expert system as a preprocessor for the annealing step. When used to choose the moves for annealing, the main function of the rule-based system is to ensure that all the trial moves satisfy the hard constraints. Many of the rules dealing with the medium and soft constraints are softened or eliminated, since reducing the cost of these constraints is done using the Metropolis update in the annealing algorithm.

Another of the modifications to the rule-based system is that while the version used in the preprocessor is completely deterministic, the version used in choosing the moves for annealing selects at random from multiple possibilities that satisfy the rules equally well. This extra freedom in choosing new schedules, plus the extra degree of randomness inherent in the annealing update, helps prevent the system from getting trapped in a local minimum before it can reach a valid schedule, which is the problem with the standard deterministic rule-based system.

To improve further on the move strategy, we can take the subset of possible move choices that we have created for each class, and choose from them probabilistically rather than randomly. There may be certain kinds of moves that are more likely to be effective, so our move strategy is to select these moves with a higher probability. For example, swapping a higher level class (e.g. graduate) with a lower level class (e.g. a first or a second year type) generally has a higher acceptance, since there is little overlap between students taking these classes. Furthermore, we have experimented with two kinds of swaps, those that only involve classes offered by the same department or college and the second, swaps between classes of different departments and colleges.

Generally, the swap methods we have taken here can be considered as heuristics for pruning the neighborhood or narrowing the search space, which provides much more efficient moves and in turn an overall improvement in the results.

4.7 Experimental Results

Our computations were done with a number of goals in mind. The main objective was to provide a schedule which satisfied all hard constraints and minimized the cost of medium and soft constraints, using real-life data sets for a large university. We also aimed to find an acceptable set of annealing parameters and move strategies for general timetabling problems of this kind, and to study the effect of using a preprocessor to provide the annealing program with a good starting point. Finally, we wanted to make a comparison of the performance of the three different cooling schedules, geometric cooling, adaptive cooling, and reheating based on cost.

We spent quite some time finding optimal values for the various parameters for the annealing schedule, such as the initial temperature, the parameters controlling the rate of cooling (α for geometric cooling, a for adaptive cooling) and reheating (K), and the number of iterations at each temperature (for more details, see Ref. [**?**]). Johnson *et al.* [**?**] noted in their SA implementation for the traveling salesman problem (TSP) that the number of steps at each temperature (or the size of the Markov chain) needed to be at least proportional to the "neighborhood" size in order to maintain a high-quality result. From our experiments we found the same to be true for the scheduling problem, even though it is very different from the TSP. Furthermore, in a few tests for one semester we fixed the number of classes and professors but varied the number of rooms and time slots, and found that the final result improves as the number of iterations in the Markov chain becomes proportional to a combination of the number of classes, rooms and time slots. We also observed the same behavior when we fixed the number of rooms and time slots but varied number of classes.

Our study case involved real scheduling data covering three semesters at Syracuse University. The size and type of the three-semester data is shown in Table 4.1. Nine types of rooms were used: auditoriums, classrooms, computer clusters, conference rooms, seminar rooms, studios, laboratories, theaters, and unspecified types. Staff and teaching assistants are considered part of the set of professors. Third semester (summer) data was much smaller than other semesters, however, there were additional space and time constraints and fewer available rooms. Our data was quite large in comparison to data used by other researchers. For example, high school data used by Peterson and colleagues [**?**, **?**] consists of approximately 1000 students, 20 different possible majors, and an overall periodic school schedule (over weeks). In the case of

Abramson *et al.* [?], their data set was created randomly and was relatively small, and they stated that problems involving more than 300 tuples were very difficult to solve.

Table 4.1 lists all major components of the data we have used. Timetabling problems can be characterized by their *sparseness*. After the required number of lessons N_l have been scheduled, there will be $N_{sp} = (N_x N_t - N_l)$ spare space-time slots, hence, the sparseness ratio of the problem is defined as the ratio $N_{sp}/(N_x N_t)$. The denser the problem, the lower the sparseness ratio, and the harder the problem is to solve. Also, for dense problems, there is an additional correlation involving the problem size. Table 4.2 shows the sparseness of the three-semester data. For university scheduling, the sparseness ratio generally decreases as the data size (particularly the number of classes) increases, so the problem becomes harder to solve. Including student preferences makes the problem much harder, but these are viewed as medium constraints and thus are not necessarily satisfied in a valid solution.

Our overall results are shown in Tables 4.3 and 4.4. These tables show the percentage of classes that could be scheduled in accordance with the hard constraints. In each case (apart from the expert system, which is purely deterministic), we have done 10 runs (with the same parameters, just different random numbers), and the tables show the average of the 10 runs, as well as the best and worst results. The MFA results are different only due to having different initial conditions. Each simulated annealing run takes about 10 to 20 hours on a Unix workstation, while a single MFA run takes approximately an hour and an expert system run takes close to two hours.

As expected, each of the methods did much better for the third (summer) semester data, which has a higher sparseness ratio. Our results also confirm what we expected for the different cooling schedules for simulated annealing, in that adaptive cooling performs better than geometric cooling, and reheating improves the result even further.

When a random initial configuration is used, simulated annealing performs very poorly, even worse than the expert system (ES). However, there is a dramatic improvement in performance when a preprocessor is used to provide a good starting point for the annealing. In that case, using the best cooling schedule of adaptive cooling with reheating as a function of cost, we are able to find a valid class schedule every time.

In the case of mean-field annealing, the overall results are generally below those of SA and ES. In addition, we have found in the implementation of this method that the results were quite sensitive to the size of the data as well the type of constraints involved. If we confine ourselves to the set of hard constraints, the results are as good as or even better than the other methods. However if we take into account the medium and soft constraints, that is, the overall cost function, this method does not perform as well.

Student preferences are included only as medium constraints in our implementation, meaning that these do not have to be satisfied for a valid solution, but they have a high priority. For the valid schedules we have produced, approximately 75% of the student preferences were satisfied. This is reasonably good (particularly since other approaches do not deal with student preferences at all), but we are working to improve upon this result.

4.8 Conclusions

We have successfully applied simulated annealing to the difficult problem of academic scheduling for a large university. Feasible schedules were obtained for real data sets, including student preferences, without requiring enormous computational effort.

Mean-field annealing works well for small scheduling problems, but does not appear to scale well to large problems with many complex constraints. For this problem, both simulated annealing and the rule-based system were more effective than MFA. It is more difficult to tune the parameters for MFA than for simulated annealing, and because of the complexity and size of the Potts neural encoding, there seems to be no clear way of preserving the state of a good initial configuration provided by a preprocessor when using MFA.

Using a preprocessor to provide a good initial state greatly improved the quality of the results for simulated annealing. In theory, using a good initial state should not be necessary, and any initial state should give a good result, however in practice, we do not have an ideal cooling schedule for annealing, or an ideal method for choosing trial moves and efficiently exploring the search space, and there are restrictions on how long the simulation can take. In general, for very hard problems with large parameter spaces that can be difficult to search efficiently, and for which very slow cooling would be much too time-consuming, we might expect that a good initial solution would be helpful. We used a fairly complex rule-based expert system for the preprocessor, however the type of preprocessor may not be crucial. Other fast heuristics could possibly be used, for example a graph coloring approach [?], or it may be possible to just utilize the schedule from the same semester for the previous year. A modified version of the rule-based system was used to choose the trial moves for the simulated annealing, and the high acceptance rate provided by this system was crucial to obtaining good results.

As expected, for the simulated annealing, adaptive cooling performed better than geometric cooling, and using reheating improved the results even further. The best results were obtained using simulated annealing with adaptive cooling and reheating as a function of cost, and with a rule-based preprocessor to provide a good initial solution. Using this method, and with careful selection of parameters and update steps, we were able to generate solutions to the class scheduling problem using real data for a large university. None of the other methods were able to provide a complete solution.

Our main conclusion from this work is that simulated annealing, with a good cooling schedule, optimized parameters, carefully selected update moves, and a good initial solution provided by a preprocessor, can be used to solve the academic scheduling problem at a large university, including student preferences. Similar approaches should prove fruitful for other difficult scheduling problems.

Table 4.1: Size of the data set for each of the three semesters.

	First Semester	Second Semester	Third Semester
Rooms	509	509	120
Classes	3839	3590	687
Professors	1190	1200	334
Students	13653	13653	2600
Buildings	43	43	11
Schools and/or Colleges	20	21	17
Departments or Course Prefixes	143	141	108
Areas of Study (majors)	200	200	200

Table 4.2: The sparseness ratios of the problem for the data sets for each of the three semesters. Lower values indicate a harder problem.

Academic Time Period	Sparseness ratio
First Semester	0.50
Second Semester	0.53
Third Semester	0.62

Table 4.3: Percentage of classes scheduled using the different methods. The averages and highest and lowest values were obtained using 10 independent runs for simulated annealing (SA) and mean-field annealing (MFA). The expert system (ES) is deterministic so the results are from a single run. *No preprocessor was used with the three methods.*

Academic Time Period	Algorithm	Scheduled (average) %	Highest Scheduled %	Lowest Scheduled %
First Semester	SA (geometric)	65.00	67.50	56.80
	SA (adaptive)	67.80	70.15	61.20
	SA (cost-based)	70.20	72.28	68.80
	ES	76.65	76.65	76.65
	MFA	65.60	71.00	61.00
Second Semester	SA (geometric)	65.65	68.00	57.10
	SA (adaptive)	68.50	70.10	60.77
	SA (cost-based)	75.14	77.68	70.82
	ES	79.00	79.00	79.00
	MFA	67.20	75.00	65.00
Third Semester	SA (geometric)	83.10	86.44	68.50
	SA (adaptive)	85.80	89.00	70.75
	SA (cost-based)	91.20	95.18	85.00
	ES	96.80	96.80	96.80
	MFA	88.00	95.00	82.00

Table 4.4: Percentage of scheduled classes, averaged over 10 runs of the same initial temperature and other parameters, for three terms using simulated annealing *with an expert system as preprocessor.*

Academic Time Period	Algorithm	Scheduled (average) %	Highest Scheduled %	Lowest Scheduled %
First Semester	SA (geometric)	93.90	95.12	85.20
	SA (adaptive)	98.80	99.20	95.00
	SA (cost-based)	100.0	100.0	100.0
Second Semester	SA (geometric)	95.00	98.95	89.40
	SA (adaptive)	99.00	99.50	98.50
	SA (cost-based)	100.0	100.0	100.0
Third Semester	SA (geometric)	97.60	98.88	90.90
	SA (adaptive)	100.0	100.0	100.0
	SA (cost-based)	100.0	100.0	100.0

Bibliography

[AD92] R. W. Ashford and R. C. Daniel. Some lessons in solving practical integer programs. *Journal of the Operational Research Society*, 43(5):425–433, 1992.

[AK89] Emile H.L. Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley, Chichester, 1989.

[AL87] Emile H. L. Aarts and Peter J. M. Van Laarhoven. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers, 1987.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[Bab92] Lászlő Babai. Combinatorial optimization is hard. *Focus*, 12(5):3–18, September 1992.

[BBI92] Robert E. Bixby, E. Andrew Boyd, and Ronni R. Indovina. MIPLIB: A test set of mixed integer programming problems. *SIAM News*, 25(20):16, March 1992.

[Bea90] J. E. Beasley. OR-library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[Ber91] Dimitri P. Bertsekas. *Linear Network Optimization: Algorithms and Codes.* MIT Press, Cambridge, MA, 1991.

[BJN+93] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Gabriele Sigismondi, and Pamela Vance. Formulating a mixed integer programming problem to improve solvability. *Operations Research*, 41(6):1013–1019, 1993.

[Bla93] Jacek Blazewicz. *Scheduling in computer and manufacturing systems.* Springer-Verlag, New York, 1993.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[Cof76] E. G. Coffman. *Computer and Job Shop Scheduling Theory.* Wiley, New York, 1976.

[Dav87] Lawrence Davis. *Genetic Algorithms and Simulated Annealing.* Morgan Kaufmann, Los Altos, CA, 1987.

[Dav91] Lawrence Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[Fis81] Marshall L. Fisher. The lagrangian relaxation method for solving integer programming problems. *Management Science*, 27(1):1–18, January 1981.

[Fou84] L. R. Foulds. *Combinatorial Optimization for Undergraduates.* Springer Verlag, New York, 1984.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[GKP92] Fred Glover, Darwin Klingman, and Nancy Phillips. *Network Models in Optimization and Their Applications in Practice.* Wiley, New York, 1992.

[Glo89] Fred Glover. Tabu search– part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[Glo90a] Fred Glover. Tabu search– part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[Glo90b] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[GLS88] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization.* Springer-Verlag, New York, 1988.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison–Wesley, Reading, MA, 1989.

[HS84] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, MD, 1984.

[Hu82] T. C. Hu. *Combinatorial Algorithms.* Addison-Wesley, Reading, MA, 1982.

[Joh92] David S. Johnson. The NP-completeness column: An ongoing guide. *Journal of Algorithms*, 13(3):502–524, September 1992.

[Jun85] Michael Junger. *Polyhedral Combinatorics and the Acyclic Subdigraph Problem.* Helderman Verlag, Berlin, 1985.

[KLS91] Bernhard Korte, László Lovász, and Rainer Schrader. *Greedoids.* Springer-Verlag, New York, 1991.

[Law76] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids.* Saunders College Publishing, Fort Worth, 1976.

[Law85] Eugene Lawler. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, New York, 1985.

[MT90] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations.* Wiley, New York, 1990.

[NW88] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley and Sons, New York, 1988.

[OLK85] M. O'hEigeartaigh, J. K. Lenstra, and A. H. G. Rinnooy Kan. *Combinatorial Optimization: Annotated Bibliographies.* Wiley, New York, 1985.

[Oxl92] J. G. Oxley. *Matroid Theory.* Oxford University Press, New York, 1992.

[Pap94] Christos H. Papadimtriou. *Computational Complexity.* Addison-Wesley, Reading, MA, 1994.

[PR88] R. Gary Parker and Ronald L. Rardin. *Discrete Optimization.* Academic Press, Boston, 1988.

[PR91] Manfred Padberg and Giovanni Rinaldi. A branch–and–cut algorithm for the resolution of large–scale symmetric travelling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[PS82] Christos H. Papadimtriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Ree93] Colin R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems.* Halsted Press, New York, 1993.

[Rei91] Gerhard Reinelt. TSPLIB— a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[Sch87] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1987.

[Sha79] Jeremy F. Shapiro. A survey of lagrangian techniques for discrete optimization. *Annals of Discrete Mathematics*, 5:113–138, 1979.

[SM89] Harvey M. Salkin and Kamlesh Mathur. *Foundations of Integer Programming.* North-Holland, New York, 1989.

[Sto92] Mechthild Stoer. *Design of Survivable Networks*, volume 1531 of *Lecture Notes in Mathematics*. Springer Verlag, Heidelberg, 1992.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA, 1983.

[Wil85] H. P. Williams. *Model Building in Mathematical Programming.* Wiley, New York, second edition, 1985.