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

by

M. A. Saleh Elmohamed

Abstract of Dissertation – First Draft

May, 2000

Abstract

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.

Combinatorial optimization problems involve choosing the best solution from a finite set of mutually exclusive outcomes and are often deceptively difficult to solve.

To summarize, we have investigated a variety of optimization algorithms, in particular, random search, hill climbing, knowledge-based systems (or rule-based expert systems), mean-field (feedback neural net-based) annealing, and simulated annealing methods to solve a large scale optimization problem, namely university academic course scheduling, represented as generalized assignment-type problems. The core heuristics of our approach are annealing-based and includes mean-field annealing, simulated annealing with three different cooling schedules, and the use of two types of preprocessors, namely rule-based and graph coloring. Moreover, preprocessors are only used with simulated annealing to provide a good starting point for the algorithm.

Along with the use of preprocessing we also have investigated a number of techniques to accelerate (i.e. 'speed up') the convergence of annealing to a good quality solution. These techniques include the selection of appropriate annealing schedules and its subsequent parameter fine-tuning, as well as recording the best solution encountered in an annealing run. Others that we have not investigated such as approximate random number generation and the implementation of a local search trigger. Also, in this thesis we have not investigated other more arduous to implement techniques such parallel, nested, and two-stage simulated annealing.

Most of these acceleration techniques are not mutually exclusive and, as our investigations have shown, that several can often be used in combination to produce time savings of a magnitude greater than those available through the use of individual

acceleration techniques.

In terms of convergence speedup as well as solution quality, 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 we investigated were able to provide a complete valid schedule. The same processing method using a graph-coloring preprocessor did extremely well but not as good as the approach with the rule-based preprocessor.

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

by

M. A. Saleh Elmohamed

M.S., Syracuse University

Dissertation

May, 2000

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

By

M. A. Saleh Elmohamed

M.S., Syracuse University

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

May, 2000

Approved ______________________________

Dr. Geoffrey Fox

Date ______________________________

Abstract

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.

Combinatorial optimization problems involve choosing the best solution from a finite set of mutually exclusive outcomes and are often deceptively difficult to solve.

To summarize, we have investigated a variety of optimization algorithms, in particular, random search, hill climbing, knowledge-based systems (or rule-based expert systems), mean-field (feedback neural net-based) annealing, and simulated annealing methods to solve a large scale optimization problem, namely university academic course scheduling, represented as generalized assignment-type problems. The core heuristics of our approach are annealing-based and includes mean-field annealing, simulated annealing with three different cooling schedules, and the use of two types of preprocessors, namely rule-based and graph coloring. Moreover, preprocessors are only used with simulated annealing to provide a good starting point for the algorithm.

Along with the use of preprocessing we also have investigated a number of techniques to accelerate (i.e. 'speed up') the convergence of annealing to a good quality solution. These techniques include the selection of appropriate annealing schedules and its subsequent parameter fine-tuning, as well as recording the best solution encountered in an annealing run. Others that we have not investigated such as approximate random number generation and the implementation of a local search trigger. Also, in this thesis we have not investigated other more arduous to implement techniques such parallel, nested, and two-stage simulated annealing.

Most of these acceleration techniques are not mutually exclusive and, as our investigations have shown, that several can often be used in combination to produce time savings of a magnitude greater than those available through the use of individual

acceleration techniques.

In terms of convergence speedup as well as solution quality, 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 we investigated were able to provide a complete valid schedule. The same processing method using a graph-coloring preprocessor did extremely well but not as good as the approach with the rule-based preprocessor.

Acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Introduction

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.

Combinatorial optimization problems involve choosing the best solution from a finite set of mutually exclusive outcomes and are often deceptively difficult to solve.

To summarize, we have investigated a variety of optimization algorithms, in particular, random search, hill climbing, knowledge-based systems (or rule-based expert systems), mean-field (feedback neural net-based) annealing, and simulated annealing methods to solve a large scale optimization problem, namely university academic course scheduling, represented as generalized assignment-type problems. The core heuristics of our approach are annealing-based and it includes mean-field annealing, simulated annealing with three different cooling schedules, and the use of two types of preprocessors, namely rule-based and graph coloring. Moreover, preprocessors are only used with simulated annealing to provide a good starting point for the algorithm.

*The terms "search problem" and "optimization problem" are considered synonymous. The search for the best feasible solution is the optimization problem.

Along with the use of preprocessing we also have investigated a number of techniques to accelerate (i.e. 'speed up') the convergence of annealing to a good quality solution. These techniques include the selection of appropriate annealing schedules and its subsequent parameter fine-tuning, as well as recording the best solution encountered in an annealing run. Others that we have not investigated such as approximate random number generation and the implementation of a local search trigger. Also, in this thesis we have not investigated other more arduous to implement techniques such parallel, nested, and two-stage simulated annealing.

Most of these acceleration techniques are not mutually exclusive and, as our investigations have shown, that several can often be used in combination to produce time savings of a magnitude greater than those available through the use of individual acceleration techniques.

In terms of convergence speedup as well as solution quality, 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 we have investigated were able to provide a complete valid schedule. The same processing method using a graph-coloring preprocessor did quite well but not as good as the approach with the rule-based preprocessor.

1.1 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. Such analysis would involve the determination of how varying the parameters of the method(s) used would affect the quality of the results.

The optimization problem we have closely tackled is university course scheduling, and 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.

1.2 Outline

Chapter 2 is an introduction to combinatorial optimization along with discussion of the inherent difficulty of many combinatorial problems, as well as look at some tailored and generally applicable heuristics. We introduce the problem of course scheduling in chapter 3, giving a definition and associated three types of constraints. Also, the chapter outlines the general form of an assignment-type problem and outlines a representation of course scheduling as a generalized assignment type.

In chapter 4 we look into the structure of the preprocessor used, namely the rule-based expert system giving a sample of of some of the rules that were used. Then we start the discussion about the core processor in chapter 5; including important discussion and analysis of energy landscapes and annealing schedules. Chapter 6 looks into our multi-phase (or hybrid) approach, discussing strategies on the choice of moves during the simulation followed by set of experimental results.

Our approach to mean field annealing is given in some details in chapter 7. Followed by discussion in chapter 8 on the use of graph coloring as a tool for course scheduling. We have used the graph-coloring approach as another preprocessor in our multi-phase approach. Finally, in chapter 9 we outline common characteristics between the two problems of airline crew scheduling and course scheduling, and discussion on performance of local search algorithms in chapter 10. We conclude with glossary chapter of definitions and terms.

Chapter 2

Combinatorial Optimization and Heuristics

Combinatorial optimization is a general class of optimizing problems restricted to a finite, discrete solution space. Combinatorial optimization problems are of great interest in many fields of research for two reasons. First, from a practical viewpoint, due to their applicability to real world situations, good answers to combinatorial problems are often vital to the success or failure of various projects. Second, from a more theoretical viewpoint, they represent an intriguing challenge due to their simplicity of statement coupled with a difficulty of solution. It is because of this difficulty in attempting to solve many combinatorial optimization problems that algorithms designed to produce solutions to such problems are forced to be heuristic in nature, i.e. are not guaranteed to return the best solution possible. When designing a heuristic algorithm, the amount of information used that is specific to the problem can vary. This categorizes algorithms into two arbitrary groupings: tailored algorithms, that use a lot of problem specific knowledge and generally applicable algorithms, that seek to minimize the amount of problem specific information used.

2.1 Optimization Algorithms

In order to solve an optimization problem such as the traveling salesperson problem (TSP)* or scheduling , we design a series of steps, or a procedure, which we can tackle a 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.

There were a number of studies looking at the issue of the distribution of the fitness or "goodness" of solutions to a given problem and the time required to find those solutions. Generally, those studies were for the idealized cases of complete algorithms which guarantee the best possible solutions (and therefore, sometimes take

*The solution of this problem or some minor variants of it, would be useful to airlines (for routing their planes at minimal cost), to telecommunication companies (for routing information at minimal cost through cables, satellites, local networks, etc.), to computer chip designers (for placing components on a chip such that connection distances are minimized for maximum operating speed).

†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.

‡See section on NP-completeness.

a very large amount of time). However, comparisons of such complete algorithms with stochastic optimization approaches in studies in *phase transitions* [HHW96, Mou84] in problem difficulty have shown empirically that the results are unchanged [GW93, GW94], and therefore extremely important to some of the issues raised in this thesis. Furthermore, such studies have shown how the time required to find good solutions by stochastic algorithms such as Simulated Annealing increases as the phase transition is approached. Hence, the phase transition results are also relevant to the search for good (not just optimal) solutions.

2.1.1 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 in this thesis fall within this category, although Mean Field annealing adds few minor requirements to this list. 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.

2.2 Local Search

Local search is meant to represent a large class of similar techniques that can be used to find a good solution from a problem. The idea is to think of the solution space as being represented by an undirected graph. That is, each possible solution is a node in the graph. An edge in the graph represent a possible move we can make between solutions. For an optimization problem such as scheduling, even if we want to, we could never hope of writing down the graph of the problem search space.

Therefore, the simple idea of local search is that we never actually try to write the whole graph down; we just move from one possible solution to a "nearby" possible solution, either for as long as we like, or until we happen to find an optimal solution.

To set up a local search algorithm, we need to have the following:

1. A set of possible solutions, which will be the vertices in our local search graph.

2. A notion of what the *neighbors* of each vertex in the graph are. For each vertex x, we will call the set of adjacent vertices $N(x)$. The neighbors must satisfy several properties: $N(x)$ must be easy to compute from x (since if we try to move from x we will need to compute the neighbors), if $y \in N(x)$ then $x \in N(y)$ (so it makes sense to represent neighbors as undirected edges), and $N(x)$ cannot be too big, or more than polynomial in the input size (so that the neighbors of a node are easy to search through).

3. A cost function, from possible solutions to the real numbers. For example, $f(s^*) : \mathcal{R} \to \mathcal{R}$.

The most basic local search algorithm (say to minimize the cost function) is easily described:

1. Pick a starting point x.
2. While there is a neighbor y of x with $f(y) < f(x)$, move to it; that is, set x to y and continue.
3. Return the final solution.

2.2.1 The Reasoning Behind Local Search

The idea behind local search is clear; if the solution we get keeps getting better and better, we should end up with a good one. Pictorially, if we "project" the state space down to a two dimensional curve, we are hoping that the picture has a sink, or a *global optimum*, and that we will quickly move toward it. See Figure 2.1.

There are two possible problems with this line of thinking. First, even if the space does look this way, we might not move quickly enough toward the right solution. Also, for any reasonably large scale optimization problem, such as scheduling, if we start with a bad solution (i.e. something with a very large cost), it will take a lot of moves to reach the minimum. Generally, however, this is not much of a problem, as long as the cost function is reasonably simple.

The more important problem is that the solution space might not look at all the way as shown in Figure 2.1. For example, our cost function might not change smoothly when we move from a state to its neighbor. Also, it may be that there are several *local optima*, in which case our local search algorithm will hone in a local optimum and get stuck. See Figure 2.2.

The second problem, that the solution space might not "look nice", is crucial, and it underscore *the importance of setting up the problem.* When we choose the possible moves between solutions — that is, when we construct the mapping that gives us the neighborhood of each node — we are setting up how local search will behave, including how the cost function will change between neighbors, and how many local

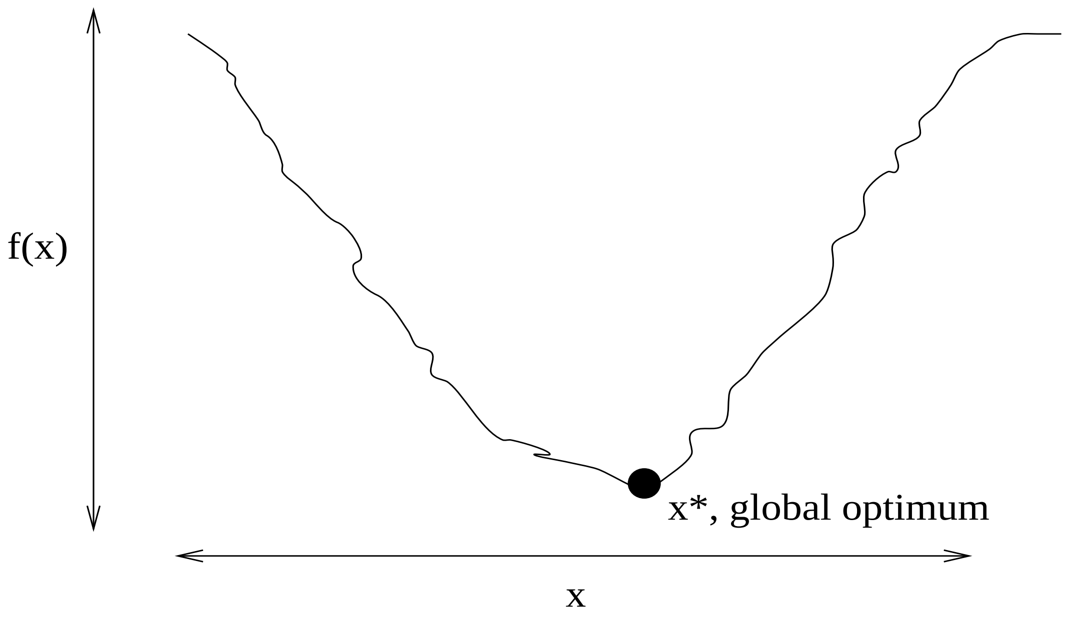

Figure 2.1: A very nice state space.

optima there are. How well local search will work depends tremendously on how smart one is in setting up the right neighborhoods, so that the solution space really does look the way we would like it to.

2.2.2 Examples of neighborhoods

- Course Scheduling [EFC96, EFC97]: A possible neighborhood structure is two assignments of a set of classes to a set of rooms over a give set of time slots, if the two assignments differ only in one class assignments. That is in the first assignment, class (A) is assigned to room (B) and class (C) is assigned to room (D); while in the second assignment, class (A) is assigned to room (D) and class (C) to room (B).

- MAX3SAT [SY91]: A possible neighborhood structure is two truth assignments

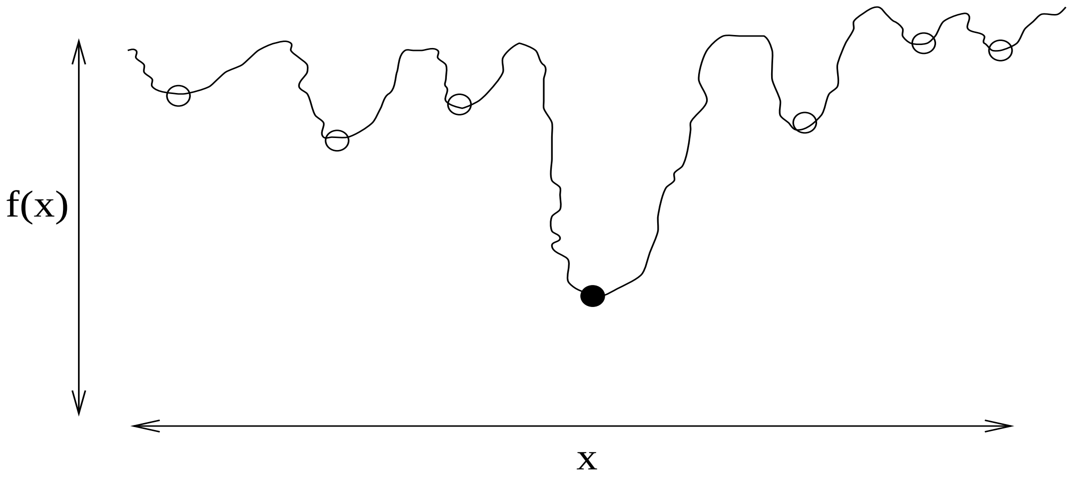

Figure 2.2: A state space with many local optima; it will be hard to find the best solution. The global optimum (x^*) is at the dark circle, while open circles represent local optima.

are neighbors if they differ in only one variable. A more extensive neighborhood could make two truth assignments neighbors if they differ in at most two variables; this trades increased flexibility for increase size in the neighborhood.

- Traveling Salesperson (TSP) [JM97]: The k-opt neighborhood of x is given by all tours that differ in at most k edges from x. In practice, using the 3-opt neighborhood seems to perform better than the 2-opt neighborhood, and using 4-opt or larger increases the neighborhood size to a point where it is inefficient.

2.2.3 Considerations to keep in mind

There are several aspects of local search algorithms that we can vary, and all can have an impact on performance. For example:

1. What are the neighborhoods $N(x)$?

2. How do we choose an initial starting point?

3. How do we choose a neighbor y to move to? (Do we take the first one we find, a random neighbor that improves f, the neighbor that improves f the most, or do we use other criteria?)

4. What if there are ties?

There are other practical things to consider. Can we re-run the algorithm several times? Can we try several of the algorithms on different machines? Issues like these can have a big impact on actual performance. However, perhaps the most important issue is to think of the right neighborhood structure to begin with; if this is right, then other issues are generally secondary, and if this is wrong, the likelihood of success is not that great.

2.2.4 Local Search Variations

There are several approaches that take advantage of search strategies in which cost-deteriorating neighbors are accepted. In the descriptions of the following search methods, we will assume we are supplied with a function 'FITNESS: $\sum \rightarrow R$', where $\sum$ is the search landscape or space, which supplies us with the solution quality. Without loss of generality, we assume higher fitnesses are better.

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 2.3):

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

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 2.3: Random Search Algorithm

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.

Hill-climbing

A simple modification of the random search strategy is to pick solutions randomly and then examine their neighborhood in the search space. Viewed graphically, in this method one would move in the search space to a vertex of lower (or possibly equal) cost. See Figure 2.4.

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

[§]The term *hill-climbing* implies a maximization problem, but the equivalent *descent* method is easily envisioned for minimization problems. So for convenience, the term hill-climbing will be used here to describe both methods without any implied loss of generality.

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 2.4: Hill Climbing Algorithm

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 would assume that the perturbation function is either defined or already given.

There are several weaknesses with hill-climbing algorithms, in general:

- They usually terminate at solutions that are only locally optimal.
- There is no information as to the amount by which the discovered local optimum deviates from the global optimum, or perhaps even other local optima.
- The optimum that's obtained depends on the initial configuration.

- In general, it is not possible to provide an upper bound for the computation time.

On the other hand, there is one alluring advantage for hill-climbing techniques: they're very easy to apply. All that's needed is the representation, the evaluation function, and a measure that defines the neighborhood around a given solution.

We also can view whether hill-climbing works well as an optimization algorithm in terms of an image of fitness landscape. That is depending on the local[¶] structure of the landscape. So, 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.

Hill-climbing methods, just like all local search methods, use an iterative improvement techniques, and it is applied to a single point – the current point – in the search space. During each iteration, a new point is selected from the neighborhood of the current point. If that new point provides a better value in light of the evaluation function, the new point becomes the current point. Otherwise, some other neighbor is selected and tested against the current point. The method terminates if no further improvement is possible, or we run out of time or patience.

It's clear that such hill-climbing methods can only provide locally optimum values, and these values depend on the selection of the starting point. Moreover, there's no general procedure for bounding the relative error with respect to the global optimum because it remains unknown. Given the problem of converging on only locally optimal solutions, we often have to start hill-climbing methods from a large variety of different starting points. The hope is that at least some of these initial locations will have a path that leads to the global optimum. We might choose the initial points at random, or on some grid or regular pattern, or even in the light of other information that's available, perhaps as a result of some prior search (e.g. based on some effort someone

[¶]Local being defined by the perturbation operator.

else made to solve the same problem). There are a few versions of hill-climbing algorithms, and differ mainly in the way a new solution is selected for comparison with the current solution.

Simulated Annealing

(See other chapters for more details) This method is based on the *Metropolis rule* which says: Pick a random neighbor, and if the cost is lower, move there. If the cost is higher, move there with some probability (that is usually set to depend on the cost differential). The idea is that possibly moving to a worse state helps avoid getting trapped at local minima.

With simulated annealing, the difference is that the probability of going to a higher cost neighbor varies with time and is gradually decreased in the course of the algorithm's execution. Lowering the *acceptance probability* is controlled by a set of parameters whose values are determined by a *cooling schedule*. Like most local search algorithms, simulated annealing is normally used as an approximation algorithm, for which much faster convergence rates are acceptable [AK89].

Tabu Search

Tabu Search (TS) based on procedures designed to cross boundaries of feasibility or local optimality, which were usually treated as barriers. It is often referred to as a meta-heuristic, and usually used in conjunction with traditional search techniques to enhance their performance. It guides a local heuristic search procedure to explore the solution space beyond local optimality.

Also, TS combines the deterministic iterative improvement algorithm with a possibility to accept cost-increasing solutions. In this way the search is directed away from local minima, such that other parts of the search space can be explored. The next solution visited is always chosen to be a *legal neighbor* of the current solution with the best cost, even if the cost is worse than that of the current solution. The set of legal neighbors is restricted by a *tabu list* designed to prevent us going back to

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 2.5: Tabu search paradigm

recently visited solutions. The tabu list is dynamically updated during the execution of the algorithm, and defines solutions that are not acceptable in the next few iterations. However, a solution on the tabu list may be accepted if its quality is in some sense good enough, in which case it is said to attain a certain *aspiration level*. See Figure 2.5.

Genetic Algorithms

(see Figure 2.6) This search strategy is based on a theme borrowed from Holland's *genetic algorithms* approach [Hol75, Gol89]. Holland uses concepts from population genetics and evolution theory to construct algorithms that try to optimize the *fitness* of a *population* of elements through *recombination* and *mutation* of their genes. There are many variations known in the literature of algorithms that follow these concepts.

One example is *genetic local search*. The general idea is as follow:

- Step 1, *initialize*. Construct an initial population of n solutions.
- Step 2, *improve*. Use local search to replace the n solutions in the population by n local optima.
- Step 3, *recombine*. Augment the population by adding m offspring solutions; the population size now equals $n + m$.
- Step 4, *improve*. Use local search to replace the m offspring solutions by m local optima.
- Step 5, *select*. Reduce the population to its original size by selecting n solutions from the current population.
- Step 6, *evolute*. Repeat Step 3 through 5 until a stop criterion is satisfied.

Evidently, *recombination* is an important step, since here one must try to take advantage of the fact that more than local optimum is available, i.e., one most exploit the structure present in the available local optima.

Generally, genetic algorithms depend greatly on the power of recombination of partial solutions. This begins with a randomly generated population of feasible solutions, then all tested against the cost function, and the solutions are kept and used to generate a new population by reproduction. The reproduction process typically involves combining features from two or more parent solutions to produce a child solution. Mutations of individual solutions are allowed. The key to a good genetic algorithm lies in the design of good reproduction rules. With effective rules, the next generation will typically contain better solutions than the previous one. The best of these solutions then reproduce to form a new generation, and so on, until a solution of acceptable quality is found, or successive generations no longer show significant improvements.

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 $\sigma = \arg\ max_{\sigma\in\Sigma}\ \{\text{Fitness}(\sigma)\}$

Figure 2.6: Genetic Algorithm

Artificial Neural Networks

Consist of networks of elementary nodes (neurons) that are linked through weighted connections. The nodes represent computational units, which are capable of performing a simple computation, consisting of a summation of the weighted inputs, followed by the addition of a constant called the threshold or bias, and the application of a nonlinear response function. The result of the computation of a unit constitutes its output. This output is used as an input for the nodes to which it is linked through an outgoing connection. The overall task of the network is to achieve a certain network configuration, for instance a required input-output relation, by means of the collective computation of the nodes. This process is often called *self-organization*. Over the years a large variety of neural network models have been proposed, which differ in the network architecture and the computational model used for self-organization. In many models, the self-organization of the network tries to find a stable network configuration. For models such as the *Hopfield network* [HT85] or the *Boltzmann machine* [AK89], the self-organization can be modeled in terms of local search. Here the issue of escaping from local optima is again of prime importance. In a number of cases it is resolved by applying probabilistic methods, as in simulated annealing. This takes us to the mean field annealing model outlined in another chapter.

2.3 Algorithms, Landscapes, and Problem Classes

There is much to be learned by examining the interaction between optimization algorithms and landscapes, both in terms of understanding the structure of landscapes, and towards improving the algorithms we use: as well as phase transitions in problem difficulty, there are many other robust and interesting structural properties which deserve further research. One useful approach to the understanding of algorithms in that context can be obtained by generating landscapes according to certain rules, and studying the statistical properties of the ensemble. So, for example, given a set of statistical properties of landscapes, one can begin to determine which of these properties the algorithms would require to perform well.

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 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.

2.4 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 optimization problems is given by the move set: a small rearrangement 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. $d(\vec{x}, \vec{x}') = 1$.

2.5 Computational Complexity

Computational complexity was developed largely in the 1970s to evaluate whether algorithms should be classified as 'easy' or 'hard'. Parker and Rardin [PR88] define the goal of complexity theory as:

...seeking to classify problems in terms of the mathematical order of the computational resources required to solve the problems via digital computer algorithms.

Combinatorial optimization has given computational complexity many motivations and insights and has served as a test-bed for many new algorithmic ideas.

2.5.1 Algorithms and Problems

An algorithm is a step-by-step procedure for solving a problem. A problem can be viewed as a domain of problem instances, with a question than can be asked about any of the instances. A problem can be described as a generic instances together with a question that can be asked about this generic formulation.

An algorithm A solves a problem P if , given any instance I of P as input data, it will generate the answer to P's question. It is known that for some problems no algorithm exists (i.e. an algorithm that will generate an answer to the problem's question for all instances). In problems like the TSP however, at least one algorithm does exist, that of complete enumeration.

When estimating the running time of an algorithm A, 'time' is expressed as a function of the problem size (in the case of the TSP this is the number of cities n), and measures 'steps' rather than the actual time taken to run the algorithm. Order notation (or O-notation) is used to express this running time. If A has a running time of $O(f(n))$ then there is some constant c such that the algorithm's running time for all instances is bounded by $cf(n)$ (complete enumeration for problems such as TSP takes time $O((n-1)!)$).

An algorithm may or may not be well behaved. It is well behaved if the number of steps required can be accurately predicted for a given problem instance, and this prediction does not vary from one instance to another with the same size. Some algorithms however are not well behaved and the number of steps required to solve problem instances of the same size can vary widely. Assigning a complexity function to a problem requires a unique 'time' value for each problem size. There are two approaches in measuring how long an algorithm takes. Either the average or the maximum number of steps can be counted over all instances of the same size. The second form of measurement, worst-case-analysis, has the disadvantage that bad performances might be very rare in practice. Average-case analysis involves choosing a probability function for the instances of the same size, a potentially difficult task. Another difficulty of average-case analysis is the dependence between the stages of many algorithms, making statistical analysis complicated. Generally, when evaluating a problem instance the average running time is not really an issue, but through worst-case analysis there is at least guarantee of how long it will take. The worst-case approach leads to the theory of NP-completeness.

2.5.2 Polynomial Time

An algorithm is considered 'good' when it is efficient enough to use in practice. Here efficiency is defined as having worst-case complexity bounded by a polynomial function of the problem size n. This splits algorithms into two distinct classes, polynomial and exponential. Although exponential algorithms may do better than polynomial algorithms for small n, exponential algorithms will perform much worse than polynomial algorithms as the problem size increases.

The precise definition of problem size can have a large bearing on the function derived for the worst-case complexity. But as long as the size is a measurement in some sense of the problem dimension, a polynomial algorithm will remain polynomial and an exponential algorithm will remain exponential, regardless of the choice of the definition[‖].

2.5.3 Characteristic of NP-Complete Problems

If we pick three different and known NP-complete problems, let say **Satisfiability**, **Knapsack**, and **Clique**, we would notice the following characteristics about them.

- Each is solvable, and a relatively *simple* approach solves it (although the approach may be time-consuming). For each of them, we can simply enumerate all the possibilities: all ways of assigning the logical values of n variables, all subsets of the set S, all subsets of n vertices in G. If there is a solution, it will appear in the enumeration of all possibilities; if there is no solution, testing all possibilities will demonstrate that.

- There are 2^n cases to consider if we use the approach of enumerating all possibilities (where n depends on the problem). Each possibility can be tested in a relatively small amount of time, so the time to test all possibilities and answer *yes* or *no* is proportional to 2^n.

- They come from different fields; logic, number theory, and graph theory, respectively.

- If it were possible to guess perfectly, we could solve each problem in relatively little time. For example, if someone could guess the correct assignment or the correct subset, we could simply verify that the formula had been satisfied or a correct sum had been determined, or a clique had been identified, etc. The verification process could be done in time bounded by a polynomial of the size of the problem.

[‖]The theory requires that the measure of input size must reflect (to within a polynomial) the actual length of a concise encoding of the problem instance as a sequence of symbols (i.e. as it would be stored in a computer).

2.5.4 The Classes P and NP

Let P be the collection of all problems for which there is a solution that runs in time bounded by a polynomial function of the size of the problem. For example, you can determine whether an item is in a list in time proportional to the size of the list (simply by examining each element in the list to determine whether it is the correct one), and you can sort all items in a list into ascending order in time bounded by the square of the number of elements in the list (using, for example, the well-known bubble sort algorithm.) There may also be faster solutions; that is not important here. Both the searching problem and the sorting problem are in P because they can be solved in time n and n^2, respectively.

For most problems, polynomial time algorithms are about the limit of feasible complexity. Any problem that could be solved in time $n^{1,000,000,000}$ would be in P, even though for large values of n, the time to perform such an algorithm might be prohibitive. Notice also that we do not have to know an explicit algorithm, we just have to be able to say that such an algorithm exists.

By contrast, let NP be the set of all problems that can be solved in time bounded by a polynomial function of the size of the problem, *assuming the ability to guess perfectly.* (in the literature, this "guess function" is called an oracle machine or a nondeterministic Turing machine.) This guessing is called nondeterminism.

Of course, no one can guess perfectly. Guessing is simulated by cloning an algorithm and applying one version of it to each possible outcome of the guess. Essentially, the idea is equivalent to a computer programming language in which IF statements could be replaced by GUESS statements: instead of testing a known condition and branching depending on the outcome of the test, the GUESS statements would cause the program to fork, following two or more paths concurrently.

The ability to guess can be useful. For example, instead of deciding whether to assign the value TRUE or FALSE to variable v_1, the nondeterministic algorithm can proceed in two directions: one assuming TRUE had been assigned to v_1, and the other assuming FALSE. As the number of variables increases so does the number of possible paths to be pursued concurrently.

Certainly every problem in P is also in NP because the guess function does not have to be invoked. There is also a class EXP, which consists of problems for which a deterministic solution exists in exponential time, c^n for some constant c. As noted earlier, every NP-complete problem has such a solution. Every problem in NP is also in EXP so P $\subseteq$ NP $\subseteq$ EXP.

2.5.5 The Meaning of NP-Completeness

The notion of NP-completeness tries to capture the "hardest" problems in the class NP. In other words, the way the class of NP-completeness is defined, if ever we were able to find a polynomial time algorithm for Q, an NP-complete problem, we can then find a polynomial time algorithm for any problem X that is in the class NP (and hence for every NP-complete problem). This follows by the definition of NP-completeness.

Stephen Cook [Coo71] showed that the satisfiability problem is NP-complete, meaning that it can represent the entire class NP. His important conclusion was that if there is a deterministic, polynomial time algorithm (one without guesses) for the satisfiability problem, then there is a deterministic, polynomial time algorithm for every problem in NP; that is, P = NP.

Richard Karp [Kar72] extended Cook's result [Coo71] by identifying several other problems, all of which shared the property that if any one of them could be solved in a deterministic manner in polynomial time, then all of them could. The knapsack and clique problems were identified by Karp. The results of Cook and Karp included the converse: if for even one of these problems (or any NP-complete problem) it could be shown that there was no deterministic algorithm that ran in polynomial time, then no deterministic algorithm could exist for any of them.

Be careful to distinguish between a problem and an instance of a problem. An instance is a specific case: one formula, one specific graph, or one particular set S. Certain simple graphs or simple formulas may have solutions that are very easy and fast to identify. A problem is more general; it is the description of all instances of a given type. For example, the formal statements of the satisfiability, knapsack, and clique sections are statements of problems because they tell what each specific instance of that problem must look like. Solving a problem requires finding one general algorithm that will solve every instance of that problem.

There are problems known to be solvable deterministically in polynomial time (P), and there are problems known not to have a polynomial time solution (EXP and beyond), so that P $\subseteq$ EXP and P $\neq$ EXP, meaning P $\subset$ EXP. The class NP fits somewhere between P and EXP: P $\subseteq$ NP $\subset$ EXP. It may be that P = NP, or that P $\neq$ NP.

The significance of Cook's result is that NP-complete problems have been studied for a long time by many different groups of people: logicians, operations research specialists, electrical engineers, number theorists, operating systems specialists, and communications engineers. If there were a practical (polynomial time) solution to any one of these problems, you would hope that someone would have found it by now. See Garey and Johnson's N-completeness catalog [GJ79].

Polynomial-time Reduction

Let Π and $\Pi^{'}$ be two problems and let A be an algorithm. We say that A is a *polynomial-time reduction* of $\Pi^{'}$ to Π if A is a polynomial-time algorithm ('solving' Σ^*), so that any allowed sequence starting with w and ending with v one has: $w \in \Pi^{'}$ if and if $v \in \Pi$. A problem Π is called NP-complete, if $\Pi \in$ NP and for each problem $\Pi^{'}$ in NP there exists a polynomial-time reduction of $\Pi^{'}$ to Π. It is not difficult to see that if Π belongs to P and there exists a polynomial-time reduction of $\Pi^{'}$ to Π, then also $\Pi^{'}$ belongs to P. It implies that if one NP-complete problem can be solved in polynomial time, then each problem in NP can be solved in polynomial time. Moreover, if Π belongs to NP, $\Pi^{'}$ is NP-complete and there exists a polynomial-time reduction of $\Pi^{'}$ to Π, then also Π is NP-complete.

2.6 Approximation Algorithms

When facing an NP-complete optimization, we may want to consider algorithms that do not produce optimum solutions, but solutions *guaranteed to be 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:

$$\frac{|opt(x) - A(x)|}{opt(x)} \leq \epsilon \tag{2.1}$$

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 $\epsilon = 0$.

2.7 More on Design of Heuristics

How in general we go about designing or just applying an existing algorithm for a particular purpose? Looking at the algorithms previously outlined in this chapter, we see that they are derived from a wide array of fields of knowledge [AL97] but share the same design goals.

For example, simulated annealing and tabu search are both designed for the purpose of escaping local optima, but differ in the methods used for achieving this goal. Tabu search usually makes uphill moves only when it is stuck in local optima, whereas simulated annealing can make uphill moves at any time. Also, SA is stochastic whereas tabu search is deterministic algorithm.

In comparison to classic algorithms such as greedy, divide and conquer, simplex method, and dynamic programming, both SA and TS work on complete solutions. In addition, SA and TS have more parameters to worry about, such as temperature, rate of reduction, size of memory, etc. than those classic methods. When dealing with methods such as SA, one would have to think not only whether the algorithm makes sense for the given problem but also how to choose the parameters of the algorithm so that it performs optimally. This is quite pervasive issue that accompanies the vast majority of algorithms that can escape local optima.

In addition to these general purpose optimization methods there are also specific heuristics, such as Kernighan-Lin method used to tackle problems such as TSP and graph 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 hopefully help us in picking the best algorithm for a given purpose.

It is known that effective search techniques provide a mechanism for balancing two apparently conflicting objectives: *exploiting* the best solutions found so far and at the same time *exploring* the search space.** Hill-climbing methods exploit the best

**This balance between exploration and exploitation was noted as early as the 1950s by the famous statistician G.E.P. Box [BV55].

available solution for possible improvement but neglect exploring a large portion of the search space S. In contrast, a random search – where points are sampled from S with equal probabilities – explore the search space thoroughly but foregoes exploiting promising regions of the space. Each search space is different and even identical spaces can appear very different under different representations and evaluation functions. So there's no way to choose a single search method that can serve well in every case.

Getting stuck in local optima is a serious problem. It's one of the main deficiencies that plague industrial applications of numerical optimization. Almost every solution to real-world problems in factory scheduling, academic scheduling, airline scheduling, demand forecasting, land management, and so forth, is at best only locally optimal.

What can we do? How can we design a search algorithm that has a chance to escape local optima, to balance exploration and exploitation, and to make the search independent from the initial configuration? There are few possibilities, and we'll discuss our approaches in the upcoming chapters, but keep in mind that the proper choices are always problem dependent. One option is to execute the chosen search algorithm for a large number of initial configurations of the problem. Moreover, it's often possible to use the results of previous trials to improve the choice of the initial configuration for the next trial. It might also be worthwhile to introduce a more complex means for generating new solutions, or enlarge the neighborhood size. It's also possible to modify the criteria for accepting transitions to new points that correspond with a negative change in the evaluation function. That is, we might want to accept a worse solution from the local neighborhood in the hope that it will eventually lead to something better.

2.8 Algorithmic paradigms

Heuristic algorithms seek good solutions to problems without providing any guarantee of optimality. Also, they generally fall into one three classes: construction algorithms, (partial) enumeration techniques, and iterative improvement methods. Construction algorithms operate by starting with an initial partial solution[††], iteratively augmenting it in an appropriate way to finally produce a full solution to the problem at hand. Enumeration techniques apply a more brute force approach to achieve their goals, searching through a list of solutions to find the best solution to the problem. Although the size of the solution space for most interesting combinatorial optimization problems prevents full enumeration from being a viable option, by excluding regions of the solution space, the list of enumerated solutions can be greatly reduced making such a partial enumeration algorithms more manageable. Iterative improvement algorithms begin with an initial solution to the problem and iteratively attempt to find a better solution by altering the current solution in some manner while (generally) maintaining feasibility. The iterative process generally terminates when no better solution can easily be obtained.

Each of the above algorithmic paradigms has strengths and weaknesses largely depending on the problem to which the technique is applied. For example there are some optimization problems where iterative improvement algorithms are hopelessly inadequate, and yet other problems where the reverse holds true.

For both exact and heuristic algorithms the distinction can be made between general and tailored algorithms. A tailored algorithm uses problem-specific information and can only be applied to a small set of problems, requiring the development of a new algorithm for each type of combinatorial optimization problem. Conversely, general algorithms are largely problem independent and will work for any number of different problem types. As with exact and heuristic algorithms, trade off is generally encountered. Either a tailored algorithm can be used, which makes use of problem structure and is comparatively quick but has limited applicability, or a general algorithm can be used, which is usually slower but has a wide potential for application. In this thesis, the emphasis is on generally applicable heuristic algorithms, particularly, annealing-based methods.

[††]A partial solution to a problem requires that a full solution be composed of a number of separate components. Taking a subset of these solution components results in a partial solution which can become a full solution to the problem if it is extended or enlarged in a suitable manner.

Chapter 3

The Course Scheduling Problem

3.1 Introduction to the Problem

A course schedule is characterized by a triplet (**t**,**g**,**a**) – or assignments, where **t** is timetabling assignment of time slots to a set of events, **g** is assignment of students to sections of the given courses, and **a** is classroom assignment to the course sections, all subject to constraints on these assignments.

The NP-complete *professors and classes* scheduling 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.

3.1.1 Constraints of the Problem

University course scheduling, like almost all practical problems pose constraints in varying degrees that need to be satisfied in order to generate feasible solution(s) to the problem. Perhaps at first we might think that constraining problems like this would make things easier – after all, we'll have a smaller search space to worry about and therefore fewer possibilities to consider. Well, that may be partially true but we need to keep in mind that to search for improved solutions we may have to be able to move from one solution to the next. Therefore, we need some kind of 'operators' that will act on feasible solutions and hopefully in turn generate new feasible solutions that are an improvement over what we have already found. It is really here where the geometry of the search space gets quite complicated as we found out dealing with a real world large scale academic scheduling problem.

To start with, for a single semester, we have to make a list of all the courses that will be offered. Then, we need a list of all students assigned to each class, and the professor assigned to each class too. Also, we need a list of all available classrooms noting the location, size, and other facilities that each offer (e.g. white boards, video projectors, lab equipments, and so forth). In addition, we would like to have a list of buildings including the actual distances between them.

Given the above data we have tackled the problem with respect to three types of constraints: hard, medium and soft.

Hard constraints are usually constraints that physically cannot be violated. They absolutely must be satisfied in order to have a feasible solution. This type of constraints 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:

- **Room Type***: Some classes, such as laboratories, require a certain type of room with the requisite facilities for the type of instruction. For example, a

*This can be considered as medium constraint.

chemistry lab must have beakers, Bunsen burners, the appropriate chemicals, safeguards, etc.)

- **Room Size**: A class cannot be assigned to a particular room unless the capacity of the room (denoted by R_c) is greater than or equal to the class enrollment (denoted by C_e). The smaller the difference between these two sizes, the better the assignment.

There is a cost associated with any violation of these constraints.

For the **Room Size**† constraint our cost function C_{rs} computes the magnitude of the difference between the room capacity R_c and the class enrollment C_e as follows:

$$C_{rs} = \begin{cases} (C_e - R_c)^2 & \text{if } C_e > R_c\,, \\ \tanh((1.0 - \frac{C_e}{R_c})\beta\gamma)(R_c - C_e) & \text{otherwise.} \end{cases} \tag{3.1}$$

In our experiments, the parameters β and γ were set to 0.4 and 0.8, respectively. When the class enrollment is larger than the room size, this function will always yield a much higher cost than the cost incurred from the assignment of a class with an enrollment that is smaller than the room size. The use of tanh function is to non-linearize the cost and not to make its increase directly proportional to the difference between room size and class enrollment.

There is also a cost assigned to any violation to the constraint that certain classes cannot be scheduled at the same time or have any overlap in the scheduled times. This is called *exclusion cost* C_x and derived as follows:

$$C_x = \begin{cases} (end(c_i) - s(c_j))Rlen(c_i, c_j) & \text{if } (s(c_j) > s(c_i)) \text{ and } (end(c_i) \geq s(c_j))\,, \\ (end(c_j) - s(c_i))Rlen(c_i, c_j) & \text{if } (s(c_j) < s(c_i)) \text{ and } (end(c_j) \geq s(c_i))\,, \\ (end(c_j) - s(c_i))Rlen(c_i, c_j) & \text{if } (s(c_j) \geq s(c_i)) \text{ and } (end(c_i) \geq end(c_j))\,, \\ (end(c_i) - s(c_j))Rlen(c_i, c_j) & \text{if } (s(c_i) \geq s(c_j)) \text{ and } (end(c_j) \geq end(c_i))\,. \end{cases} \tag{3.2}$$

where the relation between the two class durations is defined as follows:

†This can be considered as medium or soft constraint.

$$Rlen(c_i, c_j) = \begin{cases} len(c_i)/len(c_j) & \text{if } len(c_i) \geq len(c_j)\,, \\ len(c_j)/len(c_i) & \text{otherwise.} \end{cases} \tag{3.3}$$

The first and second choices of C_x take care of any partial overlaps between c_i and c_j, while the third and fourth choices take care of any complete overlaps between the two classes.

3.1.2 Strategy for Problem Solving

As mentioned above, hard constraints must satisfied to have a feasible solution. So, from what we have presented so far, we could say that any assignment that meets those hard constraints would solve our problem. This means that our task is quite similar to the satisfiability problem (SAT) [SKC96, SLM92, SK93, GJ79, CLR90]: to find an assignment of classes (as compared with Boolean variables) such that an overall evaluation function returns a value of TRUE. Anything that violates the constraints means that our evaluation function returns a value of FALSE. But this alone does not give us sufficient information to guide the search for a feasible solution.

Well, we might be able to employ a certain strategy that could provide this additional information. For example, we might judge the quality of the solution not just by whether or not it satisfies the constraints, but for those assignments that fail to meet the constraints, we could tally the number of times that the constraints are violated (for example, each time a student is assigned to two classes that meet at the same time we increase the tally). This would give us a quantitative measure of how poor our infeasible solutions were, and it might be useful in guiding us toward successively better solutions, minimizing the number of constraint violations. We could apply different operators for reassigning courses to classrooms, professors to courses, and so forth, and over time we would hope to generate a solution that met the available constraints. But then we have other types of constraints to worry about, and those are *medium* and *soft*.

Medium constraints are usually considered to be those constraints that fall into the gray area between the hard and soft constraints [EL87]. 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).
- The predecessor-successor relation between the topics is satisfied. That is, if undergraduate prerequisite courses are scheduled for the same day as their counterpart graduate courses, they should be given earlier than the graduate course (this facilitate learning foundational material prior to advanced material in the same day).[§]
- Two courses of a same topic are scheduled on two different working days.[¶]

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:

[‡] This can be considered a hard constraint.

[§] This can be considered as soft constraint.

[¶] This can be considered as soft constraint.

- Balance or spread out the lectures over the week. That is, courses that meet three times per week should preferably be assigned to Mondays, Wednesdays, and Fridays. Other assignments are not desired. Also, courses that meet twice a week should preferably be assigned to Mondays and Wednesdays or Tuesdays and Thursdays. Having these courses meet on consecutive days or with two or more days in between is not desired.
- For each student, balance the three-day (*Mon, Wed, Fri*) as well as the two-day (*Tue, Thu*) schedules.
- 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.
- Number of teachers assigned to a course in each semester should be bounded between a lower and an upper bounds on the number of sections of that course.
- Only professors can teach graduate courses.
- No professor can teach more than a specified number of graduate courses per year.
- 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.[**]

[||]This can be considered as medium or hard constraint.
[**]This can be considered as medium constraint.

- Faculty having the ideal number of teaching days.

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.

We have dealt with the distance minimization constraint in the following way. Given the various building preferences of the departments, we have constructed a matrix M_{IJ} between all the academic departments I and all the buildings involved in scheduling J. Using M_{IJ} in conjunction with the (appropriately scaled) distance between all buildings involved in the process, a final distance matrix Q is derived and directly used in the scheduling process. Also, unless otherwise stated, a department's home building is always the first preference for the department classes to be assigned.

Let B denote the set of all k buildings, D denote the set of all n departments. Also for department d_i where $1 \leq i \leq n$ the distance to all buildings B is a vector denoted by $< d_i, b_j >$ where $b_j \in B$ and $1 \leq j \leq k$. At each step of the scheduling process and according to d_i space preferences, d_i courses would be scheduled into buildings $B^{'}$ where $|B^{'}| \leq |B|$ and the distance is denoted by the vector $\ll d_i, b_j \gg$ where $b_j \in B^{'}$. In addition, let $|B^{'}| = k^{'}$.

Now the cost associated with the distance of department d_i is computed as follows:

$$C_{d_i}^{dist} = \frac{\sum_{j=1}^{k^{'}} \ll d_i, b_j \gg}{\sum_{j=1}^{k} < d_i, b_j >} \; k^{'} \tag{3.4}$$

The cost function is the *distance ratio* of department(i)'s buildings of preference to all available buildings multiplied by the number of department(i)'s preferred buildings. Notice that for a particular department the denominator of equation 3.4 stays fixed the scheduling process.

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.

- Some rooms may be declared unavailable during any time period.
- Daily limits must be specified to restrict the assignment of more than a specified number of classes per day to each professor.

3.1.3 Constraint handling in Course Scheduling – problem solving strategies

Certainly there are many more such soft constraints than those listed above. Any assignment that meets the hard constraints is feasible, but not necessarily optimal in light of the medium and soft (non-hard) constraints. Here is where the problem gets quite tricky. First, we have to quantify each of the non-hard constraints into mathematical terms so that we can evaluate any two candidate assignments and decide that one is better than the other. Next, we have to be able to modify one feasible solution and, hopefully, generate another feasible solution that better meets the non-hard constraints.

In terms of the first issue: each non-hard constraint has to be quantified. For example, consider the above first soft constraint, we could say that for each case where a solution is feasible, we could count up the number of times twice-a-week courses become separated by two or more days, or are placed on consecutive days, and use this as a penalty term. The lower the term, the better the solution. In fact, we could employ a similar approach to each of the non-hard constraints. But what would we do when we are done? We would still need an overall method for considering the degree of violation of each of these constraints. That is, we would have to determine the answers to questions such as: which is worse, scheduling a certain percentage of students to have back-to-back classes, or scheduling back-to-back classrooms at the opposite ends of the campus? each of these possible tradeoffs would have to be considered and quantified in some evaluation function, which poses quite a challenge!

Of course it is worse than that, because even after all these non-hard constraints have been quantified, we are still left with the problem of searching for the best assignment: the solution that is both feasible and minimizes our evaluation function for the non-hard constraints. Suppose we have a feasible solution, but it does not do very well with regard to the non-hard constraints. Say, we apply some variation operators (i.e swapping rooms or classes) to this solution and significantly improve the situation with respect to the soft constraints, but in so doing, we generate a solution that violates one hard constraint. Now what? We might choose to discard the solution since it is infeasible, or we might see if we can repair it to generate a feasible solution that still handles the non-hard constraints well. Either way, this typically a difficult task. It would be even better to come up with variation operators that never corrupt a feasible solution into an infeasible solution while still searching vigorously over the space of feasible solutions to find those that best handle the non-hard constraints, or the soft constraints in particular.

Overall, as we found out this is one of the most challenging real-world constrained problems to handle. We have tackled it through a multi-phase heuristic approach. It successfully produced legal schedules, including those of students, satisfying the hard as well as the medium and most of the soft constraints.

3.1.4 Cost Function

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. We consider the following as part of the cost function:

- *Class cost* is the sum of costs incurred from scheduling a class in a particular time period, including having a certain class and its lab scheduled in the same time period; spreading classes over the week and within each of the weekdays; and gluing to fill in holes in the timetable and allow contiguous time periods for a class that requests them.

- *Professor cost* is the sum of costs associated with each professor, including teaching two or more classes that overlap in time; being assigned more classes than the maximum number allowed to be taught; and not having a long enough time interval between classes taught by the same professor.

- *Student cost* is the sum of costs associated with each student, including clashes in class times; not having the classes evenly spread out over the week; having to schedule a class from the list of alternates rather than the list of first preferences; or classes that do not satisfy eligibility criteria.

- *Room cost* is the sum of costs associated with each classroom, including the distance cost obtained from the distance constraint; not being able to schedule a room of the specified type; and the room size cost obtained from the room constraint.

3.2 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.

The Assignment-Type Problem (ATP) [FHL96, FL92] can be summarized as follows:

Given n items and m resources, denote by c_{ij} the cost of assigning (scheduling) item i to resource j. The problem is to determine an optimal assignment of items to resources minimizing the total cost and satisfying K additional side constraints.

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

(ATP)

$$\text{Min} \quad F(x) \tag{3.5}$$

subject to

$$G_k(x) \leq 0 \qquad 1 \leq k \leq K \tag{3.6}$$

and

$$x \in X(1) \tag{3.7}$$

where

$$X(1) = \{x : \sum_{j \in J_i} x_{ij} = 1, 1 \leq i \leq n; x_{ij} = 0 \quad \text{or} \quad 1,\ 1 \leq i \leq b, j \in J_i\} \tag{3.8}$$

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:

$$x_{ij} = \begin{cases} 1 & \text{if item } i \text{ assigned to resource } j \\ 0 & \text{otherwise.} \end{cases} \tag{3.9}$$

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)

$$\text{Min} \;\; F(x) \tag{3.10}$$

subject to

$$G_k(x) \leq 0 \qquad 1 \leq k \leq K \tag{3.11}$$

and

$$x \in X(a) \tag{3.12}$$

where

$$X(a) = \{x : \sum_{j \in J_i} x_{ij} = a_i, 1 \leq i \leq n; x_{ij} = 0 \quad \text{or} \quad 1,\ 1 \leq i \leq n, j \in J_i\} \tag{3.13}$$

and

$$a = [a_1, a_2, \ldots, a_n]^T \tag{3.14}$$

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.

$$V_k(x) = \text{ Max } \{0, G_k(x)\}. \tag{3.15}$$

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

$$V(x) = \sum_{k=1}^{K} \beta_k V_k(x) \tag{3.16}$$

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)

$$Min_{x \in X(a)} P(x) = \alpha F(x) + V(x) \tag{3.17}$$

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.

3.2.1 Representation of Course Scheduling as GATP

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 [AF89], 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 $J_i = \{1, 2, \ldots, m\}$:

$$x_{ij} = \begin{cases} 1 & \text{if lecture } i \text{ starts at } j \\ 0 & \text{otherwise.} \end{cases} \tag{3.18}$$

The objective function accounts for the lecturer preferences:

$$F(x) = \sum_{i=1}^{n} \sum_{j \in J_i} c_{ij} x_{ij} \tag{3.19}$$

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:

$$x_{ij} x_{kl} \leq 0 \qquad (k, l) \in \Gamma_{ij} \ , \ 1 \leq i \leq n \ , \ j \in J_i \ . \tag{3.20}$$

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:

$$\sum_{(i,j) \in K_{bt}} x_{ij} - U_b \leq 0 \qquad 1 \leq b \leq B \ , \ 1 \leq t \leq T \tag{3.21}$$

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

The course timetabling problem can be summarized as follows:

$$Min \sum_{i=1}^{n} \sum_{j \in J_i} c_{ij} x_{ij} \ . \tag{3.22}$$

Subject to:

$$x_{ij} x_{kl} \leq 0 \qquad (k,l) \in \Gamma_{ij}, \quad 1 \leq i \leq n \ , \ j \in J_i \tag{3.23}$$

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:

$$V_{kl}^{ij}(x) = Max\{0, x_{ij} x_{kl}\} = x_{ij} x_{kl} \ . \tag{3.24}$$

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

$$V_{bt}(x) = Max\{0, \sum_{(i,j) \in K_{bt}} x_{ij} - U_b\} \ . \tag{3.25}$$

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

$$\beta_{kl}^{ij} = \gamma_{ik} L_{ijkl} \ . \tag{3.26}$$

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:

$$Min_{x\in X(1)} \sum_{i=1}^{n} \sum_{j\in J_i} \left[\alpha c_{ij}x_{ij} + \sum_{(k,l)\in\Gamma_{ij}} \gamma_{ik}L_{ijkl}x_{ij}x_{kl} \right] + \rho \sum_{b=1}^{B} \sum_{t=1}^{T} Max \left\{ 0, \sum_{(i,j)\in K_{bt}} x_{ij} - U_b \right\} . \tag{3.27}$$

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.

Chapter 4

The Rule-Based (Expert) Systems

4.1 Introduction

We have implemented a fairly complex rule-based expert system for solving the course 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 tackling the three assignment-type problems (**t**, **g**, **a**) of *timetabling*, *grouping*, and *class assignment* 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.

4.2 Examples of Rule-based Constraints

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.

4.2.1 Time and Space

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.

[If more than a room having an identical availability time are present in the database, then choose the one with the minimal magnitude of the difference between room(capacity) and course(space-requested)].

Input: CPS-615 requests space for 75 students and meeting hours of [MWF 10:40 - 11:35 am].

In the database we have, for example, the following rooms:

- room: 1-218 CST , capacity: 45 students, and pre-occupied hours: none.
- room: 207 HL, capacity 120 students, and pre-occupied hours: [MW 09:45 - 11:10 am].
- room: Kitt-Aud HBC, capacity: 127 students, and pre-occupied hours: [TTH 10:40 - 12:00 am].

Output: Room Kitt-Aud is assigned to CPS-615.

4.2.2 Type of Room

IF [room(capacity) > course(space-requested)] and [no time conflict in assigning this room] and [type(room) matches type(requested)] THEN assign the room to the course: [If identical rooms in type are present in the database, then choose the one with the minimal magnitude of the difference between room(capacity) and course(space-requested)].

Input: BIO-615 requests space (not biology lab) for 35 students and meeting hours in [MW 10:40 - 11:35].

In the database we have, for example, the following rooms:

- room: 1-218 CST, capacity: 45 students, type: classroom, and pre-occupied hours: none.

- room: 124-SIMS, capacity: 40 students, type: laboratory, and pre-occupied hours: none.

Output: Room 124-SIMS is assigned to BIO-615.

4.2.3 Room Priority

Rooms are classified as:

1. General Purpose: used by all users.
2. Priority Rooms: are used based on certain priority of the class or the department.
3. Special Purpose: are only assigned to specific users.

- *IF [room(priority) == general] and [room(capacity) > course(space-requested)] and [no time conflict in assigning this room] and [type(room) matches type(requested)]*

  THEN assign any user to this room.

- *IF [room has a priority user] and [the user is already assigned] and [room(capacity) > course(space-requested)] and [no time conflict in assigning this room] and [type(room) matches type(requested)]*

  THEN assign other users.

- *IF [room(priority) == special purpose] and [user belongs to the special users of this room] and [room(capacity) > course(space-requested)] and [no time conflict in assigning this room] and [type(room) matches type(requested)]*

  THEN assign the room to the user.

- *IF [room has a priority user] and [the user is already assigned] and [room(capacity) > course(space-requested)] and [no time conflict in assigning this room] and [type(room) matches type(requested)]*

  THEN assign other users.

- *IF [room(priority) == special purpose] and [user belongs to the special users of this room] and [room(capacity) > course(space-requested)] and [no time conflict in assigning this room] and [type(room) matches type(requested)]*

  THEN assign the room to the user.

4.2.4 Forward and Backward Rules

Our expert system also make use of the forward and the backward chaining approach of Petrie *et al.* [PCSD89]. This approach is quite suitable for planning/assignment-type problems such as course scheduling. The problem's model can be viewed in a framework as one where possible interpretations for each state (i.e. teacher) are already specified at the outset as a constrained ordered set of courses (the course database). Therefore, in order to generate a solution, the problem-solver uses a generate-and-test strategy based on this ordering.

An example to illustrate how forward/backward chaining works can be considered given the following scenario: Suppose Johnson wants to teach physics 300 (phy300) in the fall, and he has alternatives to phy300 in the fall, namely phy310 and phy318. Suppose that the maximum allowable sections of phy300 have already been allocated for fall, and suppose Johnson is now up for consideration (Richards has already been considered for phy300). The rules will try to assign phy300 to Johnson in the fall but fail becuase the maximum constraint has been satisfied for the semester.

4.2.5 The Rationale of Using a Preprocessor

- It is a good practice when using heuristics such as simulated annealing, to start at a good point in the search space or with a partial solution of the problem; regardless of the difficulty of the problem.

- A good starting point can only have a positive effect on the overall solution, and that is improving its quality.

- A good starting point or a partial solution will also have a positive effect on the convergence of the system; faster convergence to the final solution.

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.

Chapter 5

Simulated Annealing and Energy Landscapes

Simulated annealing (SA) – also known as Monte Carlo annealing, statistical cooling, probabilistic hill-climbing, stochastic relaxation, and probabilistic exchange algorithm – has been widely used for tackling different combinatorial optimization problems, particularly academic scheduling [TD95, Abr91, Vid93]. The basic algorithm is outlined in Figure 5.3.

As with any search algorithm, simulated annealing requires the answers for the following problem-specific questions:

- What is a solution?
- What are the neighbors of a solution?
- What is the cost of a solution?
- How do we determine the initial solution?

These answers yield the structure of the search space together with the definition of a neighborhood, the evaluation function, and the initial starting point. Note, however, that simulated annealing also requires answers for additional questions:

- How do we determine the initial temperature T?
- How do we determine the cooling ratio of T (or the cooling schedule)?
- How do we determine the length of each homogeneous Markov chain?
- How do we determine the halting criterion?

Before talking about possible answers to these questions, first we would like to briefly outline our intuition about annealing. It is logically clear to us that the success of annealing in many ways depends on both the cost function to which it applied and the move set used. First, imagine the move set used is that of the complete graph. Then by randomly generating moves, annealing is randomly generating states. At the least, it could keep the best random state found so far; even that would be a terrible algorithm. But if the temperature is nonzero it does still worse, since it may relinquish an optimal state. On the other hand, if the move set is terribly sparse, it may simply be impossible to get from a bad state to a good one in a reasonable number of moves. Finally, imagine that a "good" move set has been chosen, but energies have been randomly assigned to the states. In that case no information is known about a state before it is visited, and the only way to find a particularly good state is ti visit many states.

Note that this is not the case for, say, a smooth energy function. For a smooth energy function, bounds or expectations of all nearby states are known from the current state alone, and it might be possible to find a good state by visiting only a small fraction of the set of all states. Of course, "smooth" in this context means that the energy changes relatively little from move to move. So, for annealing to work well, the energies and the move set must be well matched to one another. Therefore, the key question to ask is that, intuitively, what properties must the landscape have if annealing is to be efficient?

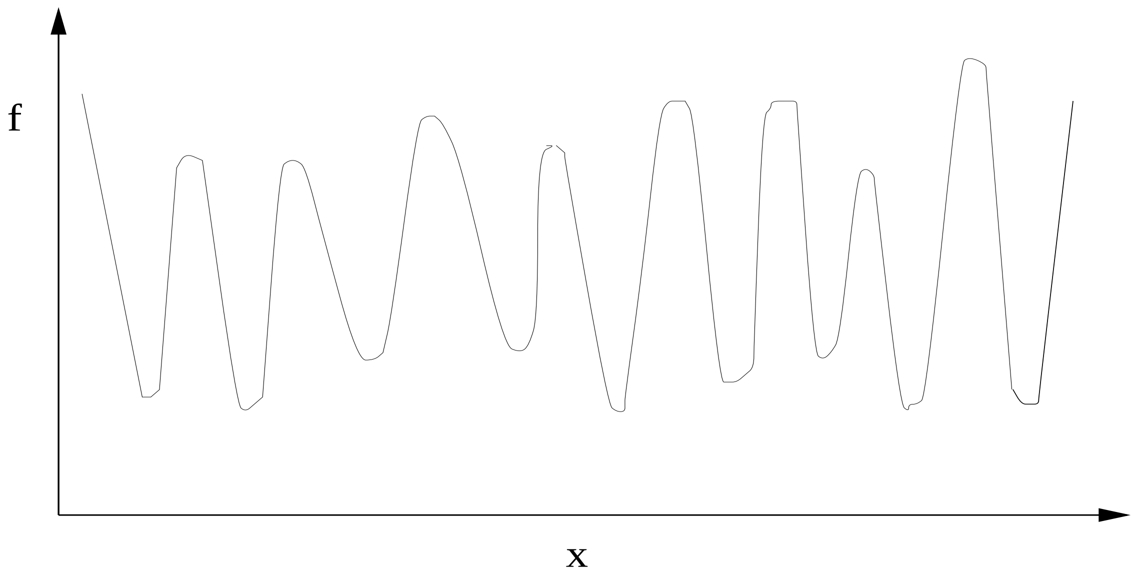

Figure 5.1: A landscape that is bad for annealing.

5.1 Good and Bad Energy Landscapes

Suppose that the energy landscape is as sketched in the "bad" landscape of Figure 5.1. States are represented along the x axis, with adjacent states being neighbors. On this landscape, the energy differences between the low-energy states (the valley bottom) are fairly small, while the energy barriers separating them (the mountains) are large. It is well known that the time required to cross a barrier of height h at temperature T is exponential in h/T, while the probability of a state of energy f is exponential in $-f/T$. So crossing the high barriers in reasonable time demands T to be large, while favoring the better valleys over the less good ones requires T to be small, implying that annealing cannot work both well and quickly on this space.

On the other hand, annealing should work well on a function like that of the "good" landscape of Figure 5.2. In this case, annealing can work in a hierarchical

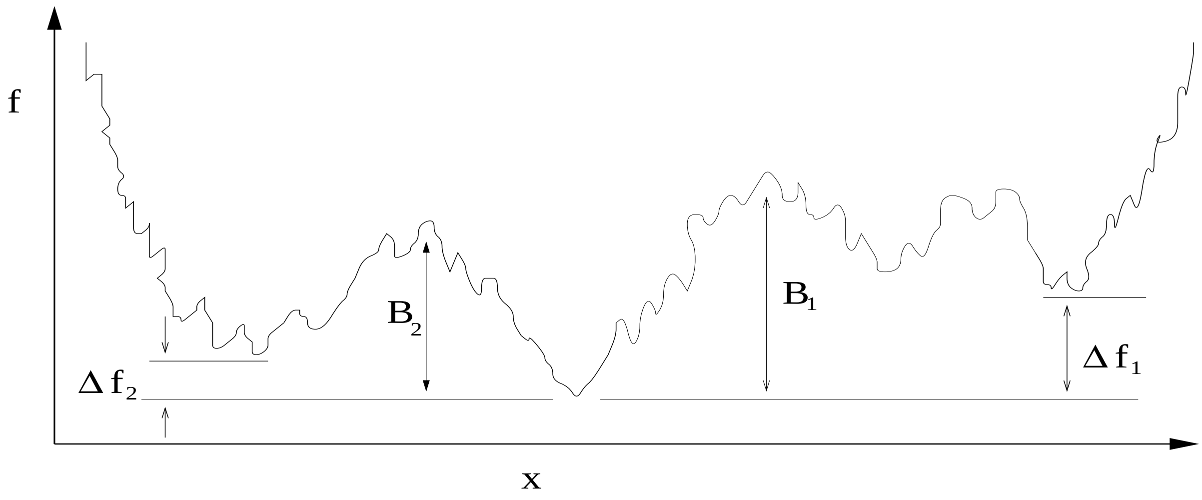

Figure 5.2: A landscape that is good for annealing.

fashion. Initially, the goal can be just to choose the better (left) of the two valleys separated by the tallest barrier B_1. While B_1 is large, the overall energy difference Δf_1 of the two valleys is also fairly large, so with a comparably sized value of T_1 we can be in the lower-energy valley with high probability in short time.

Next we can aim to settle in the better (right) of the two valleys separated by the smaller barrier B_2. While the energy difference Δf_2 between these valleys is smaller that Δf_1 was, B_2 is also smaller than B_1 and so using T_2 similarly smaller than T_1 again allows us to be in the lower-energy valley with high probability in reasonable time. This process is repeated for smaller energy scales.

The success of annealing relies on the overall energy difference of collection of states being large compared with the barriers dividing these collections. So we would conjecture that either all the energy barriers and energy differences are of the same scale (which really does not seem to be the case in practice), or else in smaller and smaller areas of the landscape the energy barriers must scale down along with the energy differences, giving a general "self-similarity" or "fractalness" like that of the "good" landscape of Figure 5.2.

5.2 On the Application of Annealing

Depending on the problem tackled, we observed that how the cooling schedule is chosen plays a major part in shaping the quality of the results obtained for academic scheduling. Initially, we 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 [KGV83, Kir84] and [ADK00].

A comprehensive discussion of the theoretical and practical details of SA is given in [AKvL97, OvG89, vLA87]. 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 5.3, 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 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

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 5.3: The Simulated Annealing Algorithm

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.

Using the search space (or the landscape) metaphor, an intuitive picture on how it works is as follows: the shape of the landscape is randomly smoothed or blurred at high temperatures, so that local peaks don't persist for too long. The solution can easily move to the fitter parts of the landscape. As the temperature is gradually reduced, the peaks begin to stick together, but the solution has escaped from their clutches and converges gradually to the top of the highest (or a high) peak.

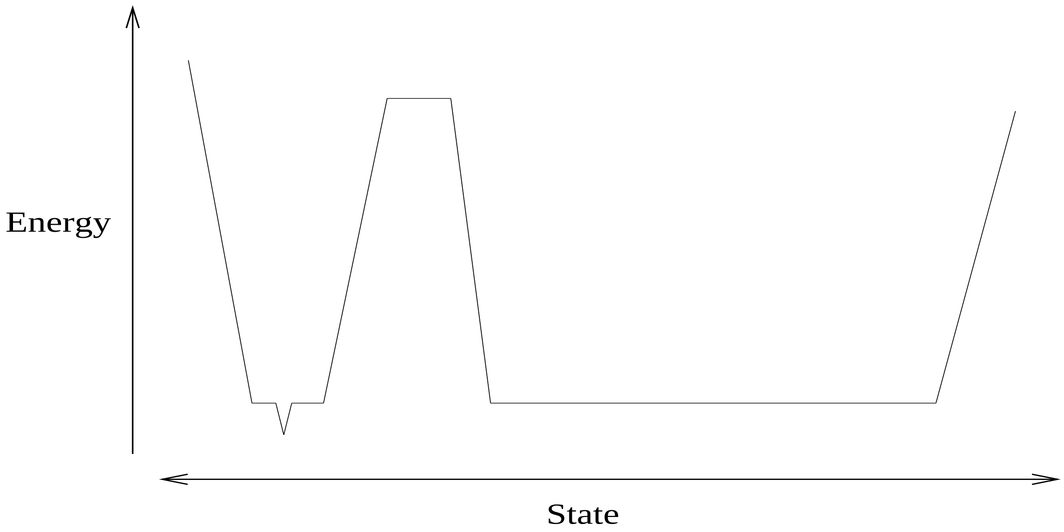

Figure 5.4: Illustrating a problem for which annealing won't work very well. The state here is a one-dimensional, with the energy of a state as shown. At low temperature an energy barrier confines the simulation for long periods of time to either the small region on the left or the larger region on the right. The small region contains the global minimum, but annealing will likely direct the simulation to the larger region, since the tiny region in the vicinity of the global minimum has negligible influence on the distribution at the high temperatures where the energy barrier can easily be traversed. (See also Figure 5.1)

As shown in Figure 5.4, the distribution at high temperature is dominated by the size of a two-energy regions, not by the depths of their lowest points. Since the global minimum happens to be in the smaller region, annealing will in this case direct the simulation away from the region where the equilibrium distribution will be concentrated at very low temperatures. To avoid such a situation we would need to be very careful about the kind of annealing schedule we choose.

In general, depending on the problem to which it is applied, SA has been shown to be quite competitive with many of the best heuristics, as shown in the work of Johnson *et al.* [JM97].

5.3 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.

5.4 Interpretation of the Solution Space

After setting up or mapping the problem we need to determine the following:

- A *feasible solution* is defined as a schedule that satisfies a subset (C') of constraints (C) : $C' \subseteq C$; provided that all hard constraints are part of C' .
- A *neighboring solution* is obtained by modifying the current solution space (i.e. swapping two or more classes in time).

- Here are two guidelines in determining whether a given constraint to be included in C' or not:

  1. A class should never be assigned to a time period if such an assignment induces only class schedules that do not satisfy all the constraints (specifically the hard constraints).
  2. A conflict between two or more classes should not be a sufficient condition for hindering an assignment.
  3. If all courses last one time period, this is equivalent to changing the color of a vertex, for example.

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.

5.5 The Annealing Schedules

The search of efficient cooling schedules may be thought of in two quite different ways, which could be called "off-line" versus "on-line", or "oracular" versus "constructive". In all cases, of course, what is desired is a cooling schedule which is as efficient as possible: one which within a specified run time produces the minimum possible expected energy, or one which produces a specified expected energy in the minimum possible run time.

In an off-line or oracular construction, the computational cost of finding the schedule itself is neglected. The extreme example of this approach is the work of Strenski and Kirkpatrick [SK91], where (via a complete expansion of the transition matrices as a function of temperature) the expected energy is computed as a function of the temperature sequence $T_1, \ldots, T_t$ (one move at each temperature).

The on-line or constructive approach is the one more commonly taken. Here, all quantities of interest are estimated during an annealing run, or with a limited amount of precomputation. For example, it is common to try to estimate the mean and variance of the energy in equilibrium at temperature T from the energies of the states produced during annealing at temperature T.

In practice it is the second approach (on-line) that is usually most relevant. On the other hand, the first approach shows what is possible, providing a yardstick against which on-line approaches may be compared. Furthermore, on-line construction of a good cooling schedule cannot proceed blindly. Off-line studies of optimal cooling schedules, whether theoretical or empirical, can indicate the qualities that a good schedule should have; it is then the job of the on-line algorithm to produce as good as possible an approximation to this ideal.

In short, let K be the number of "generations" used for annealing, t_k is the number of attempted moves made (or "time spent") in generation k, and T_k is the temperature used in that generation. Then the sequence $\{(T_k, t_k)\}_{k=1}^{K}$ and the value K are referred to as the annealing schedule.

5.5.1 Cooling Parameters

Each cooling takes a set of parameters:

1. the temperature reduction factor,
2. time at each temperature,
3. initial temperature, and
4. final temperature,

all of which must be controlled for.

In practice, among these parameters, the one with the least significance is the temperature reduction factor. (The experiments showed that if it is replaced with, say, its square root, and the number of moves at the initial temperature is doubled, the outcome will be the same.)

Since we desire a comparison of the cooling schedules over a range of run times and solution qualities, the time spent at each temperature was left as an independent variable.

The choice of initial temperature poses more of a challenge for non-geometric schedules as we will see in the next sections. Also, for non-geometric schedules the stopping criterion would vary, greatly depended on the structure of the schedule. For geometric cooling, the standard stopping criterion is to terminate the run when no moves had been accepted for some predetermined number of generations. There are also other stopping criteria.

5.5.2 Dependence on Initial Temperature

The initial temperature, T_1, is generally chosen to be high enough that the canonical distribution is close to uniform. When the Metropolis algorithm is used with a fixed distribution for candidate moves, a related criterion is that the temperature be high enough that the rejection rate is very low.

For T_1, we have simply used the upper bound suggested by White [Whi84], namely the standard deviation of the energies of randomly-selected states in the space. As indicated by Sorkin [Sor91], this can be found with sufficient accuracy with a short run at infinite temperature. For further discussion of the run lengths needed and the accuracy of the estimates produced, the reader is referred to [OvG89].

Interpretation, and identification of a good initial temperature

Based on the work of White [Whi84], Sorkin [Sor91] adopted an approach in which it was pointed out that the "cooling curves" (graphs of the mean energy at each temperature during annealing, plotted against the temperature) "quench out" at different temperatures depending on how many moves are made at each temperature. His study showed cooling curves for runs with 2^{11} and 2^{13} moves per temperature, and a temperature cooling factor of 0.9. It showed a divergence of the curves at a temperature of about 6, so at least one process (presumably the shorter-time one) was going out of equilibrium there. In another figure, he showed the relative behavior more clearly by subtracting the longer-time curve from the shorter one. (see Figures 11.5 and 11.6 in [Sor91])

The interpretation of what is happening there is that, at high temperatures both processes are in equilibrium, and so the energy difference between two states at a given instant is a random number symmetric about 0. The difference at a later instant is also random, but if the times are close together, the two difference will be correlated. The time

5.6 Our Experimental Schedules

Three annealing schedules have been used in our experiments to update the temperature of the SA algorithm in Figure 5.3: 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

$$T^{'} = \alpha T \ , \tag{5.1}$$

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. Its main disadvantage is that it doesn't take account of the state of the system in any way, and thus cannot adapt the intensity of the search depending on the difficulty of the problem.

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.

It was noted by Sorkin [Sor91] that if the same amount of time was spent at each temperature, then subproblems of all energy scales would be solved equally well. But subproblems of larger energy scale will have a larger effect on the final cost. Therefore, it is sensible tp spend somewhat more time solving these large-energy problems well, spending less time in the less important problems of smaller energy scales. We can deduce from this statement that an optimal cooling schedule should not be geometric, spending equal time at each temperature, but adaptive in nature, spending less time at lower temperatures.

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.* [ADK00], but the idea behind it are due to Kirkpatrick *et al.* [KGV83, Kir84] and White [Whi84]. 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 energy (or cost) 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 [HHW96, Mou84] can be observed as sub-parts of the problem are resolved. In some of the work dealing with the traveling salesman problem using annealing [Lis93], 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 [ADK00], 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 [ADK00, vLA87, OvG89].

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:

$$P_i(T) = \frac{e^{\frac{-C_i}{kT}}}{\sum_j e^{\frac{-C_j}{kT}}} \,. \tag{5.2}$$

The average cost is computed as:

$$< C(T) >= \sum_{i \in \mathcal{C}} C_i P_i(T) \,. \tag{5.3}$$

Therefore, the average square cost is:

$$< C^2(T) >= \sum_{i \in \mathcal{C}} C_i^2 P_i(T) \,. \tag{5.4}$$

The variance of the cost is:

$$\sigma^2(T) =< C^2(T) > - < C(T) >^2 \ . \tag{5.5}$$

Now, the specific heat is defined as:

$$C_H(T) = \frac{\sigma^2(T)}{T^2} \,. \tag{5.6}$$

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

$$T = K \cdot C_b + T(C_H^{max}) \,, \tag{5.7}$$

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 5.7), 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.* [HRSV86], 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 5.5, the new temperature T_{j+1} is computed as follows:

$$T_{j+1} = T_j \cdot e^{-\frac{aT_j}{\bar{\sigma}(T_j)}}, \tag{5.8}$$

where a is a tunable parameter. Following suggestions by Otten and van Ginneken [OvG89] and Diekmann *et al.* [DLS93], $\sigma(T_j)$ 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 $\bar{\sigma}$:

$$\bar{\sigma}(T_{j+1}) = (1-\omega)\sigma(T_{j+1}) + \omega\sigma(T_j)\frac{T_{j+1}}{T_j} \tag{5.9}$$

and set ω to 0.95. This smoothing function is used because it follows (from the form of the Boltzmann distribution, see [Sor91, Whi84]) that it preserves the key relationship:

$$\frac{d}{dT}C(T) = \frac{\bar{\sigma}^2(T)}{T^2} = C_H \tag{5.10}$$

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

5.6.1 Comments

It is unfortunate that in most published studies of automatic cooling schedules we are aware of, the actual form of the schedule is not presented. That is, while an algorithm for setting the temperatures and times is given, the resulting empirical dependence of temperature on time is not, and it is impossible to reconstruct this without re-implementing the algorithm. There is much to be learned about these schedules. For start, answers to these questions could be used to improve the scheduling algorithm: Is the schedule geometric? Could it start at a lower temperature? Should it stop at a higher temperature? How does the form change as parameters are altered?

Studies of cooling schedules also generally compare two or three algorithms – or rather, algorithm implementations – and it is hard to judge any of them in an absolute sense. Since geometric schedules were the *de facto* standard before good adaptive schedules were developed; since a number of studies, and reasoning going back to the work of White [Whi84], gives intuitive support for their efficiency, and since some adaptive schedules may wind up being roughly geometric anyway; comparisons with "ideal" geometric schedules, found by exhaustive search or any other means, would be something quite useful to have.

5.7 Annealing in Comparison with Other Algorithms

While it is valuable to know about the time versus quality tradeoff of annealing itself, knowing whether annealing is a good algorithm to use depends on its relative performance compared with other algorithms. The most significant set of experimental comparsions is that of [JAMS89, JAMS91, JM97], where for each of a number of problems annealing was compared against the best algorithm known for that problem. Generally speaking the specialized algorithms tailored to the particular problem at hand ran far faster than annealing, but annealing's results were often as good or better. Overall, if annealing was not always the best algorithm, it was at least competitive.

Chapter 6

Multi-Phase Approach to Scheduling

6.1 Our Method

Figure 6.1 illustrates the multi-phase approach we have taken to tackle a large scale course scheduling problem. The second phase of this method can be viewed as an improvement phase to the quality of the initial schedule produced in the first phase. This is accomplished by solving three different assignment-type problems (ATPs), namely, timetabling, grouping, and the classroom assignment. The first ATP tries to decrease the total number of overlapping situations by modifying the starting times of the lectures. The second ATP improves the grouping of the students by moving them from one course section to another section of the same course, with an objective of minimizing the total number of overlapping situations involving students. The third ATP improves the classroom assignment.

6.1.1 Attributes of the Data

Briefly outlining the varies attributes of the data involved in the scheduling problem.

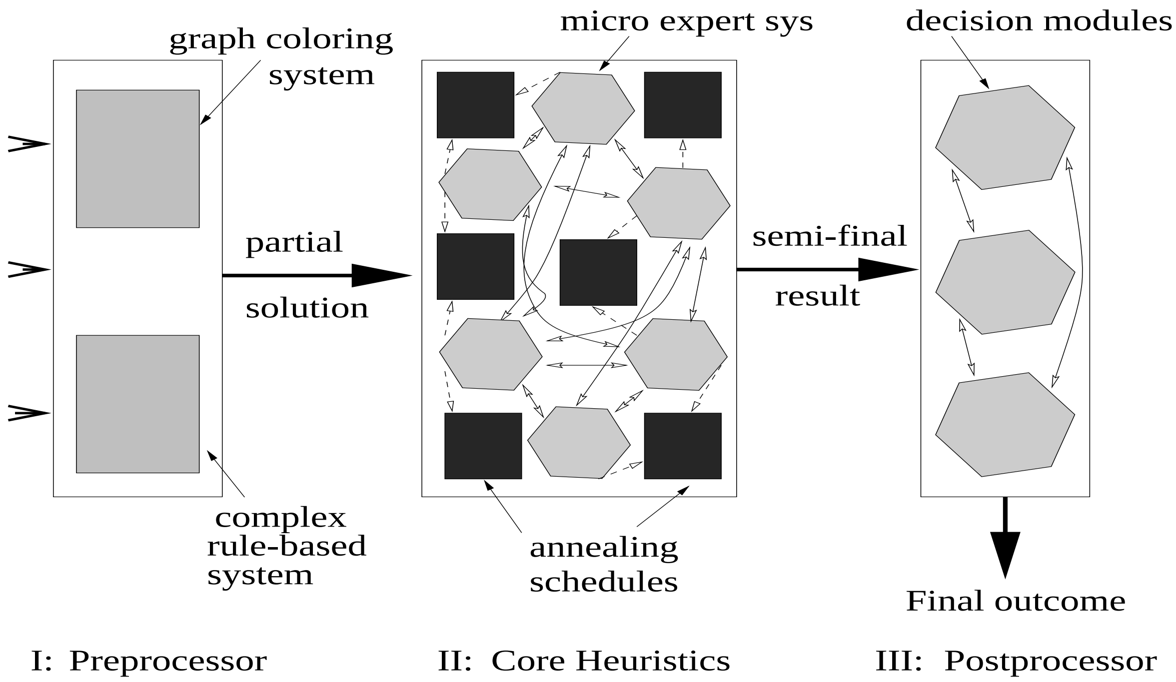

Figure 6.1: A schematic view of the system three-phase architecture.

Room Attributes

Scheduling Domains

Start Times

Global Parameters

Campuses, Buildings and Rooms

Day-Time Combinations

Common Start Times

Faculty

Courses

Campus Distances

Distances between main campus and other branches, if any exists.

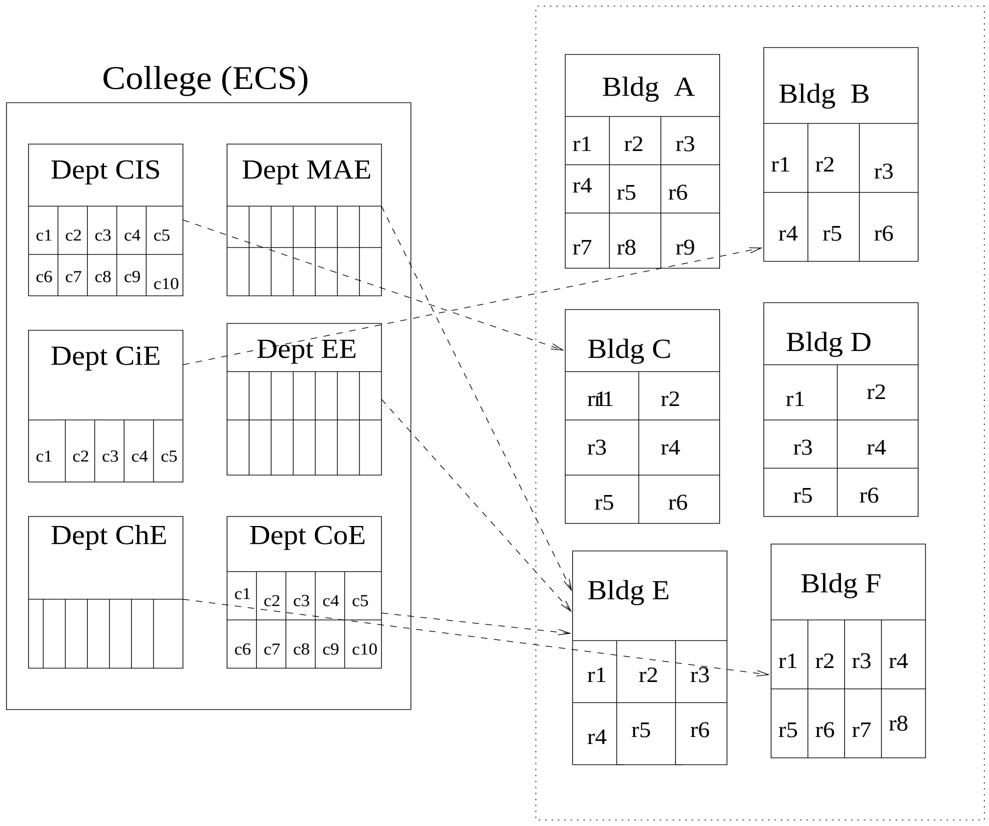

Figure 6.2: A schematic view of some of the main components of the system, colleges with their associated departments and a list of buildings.

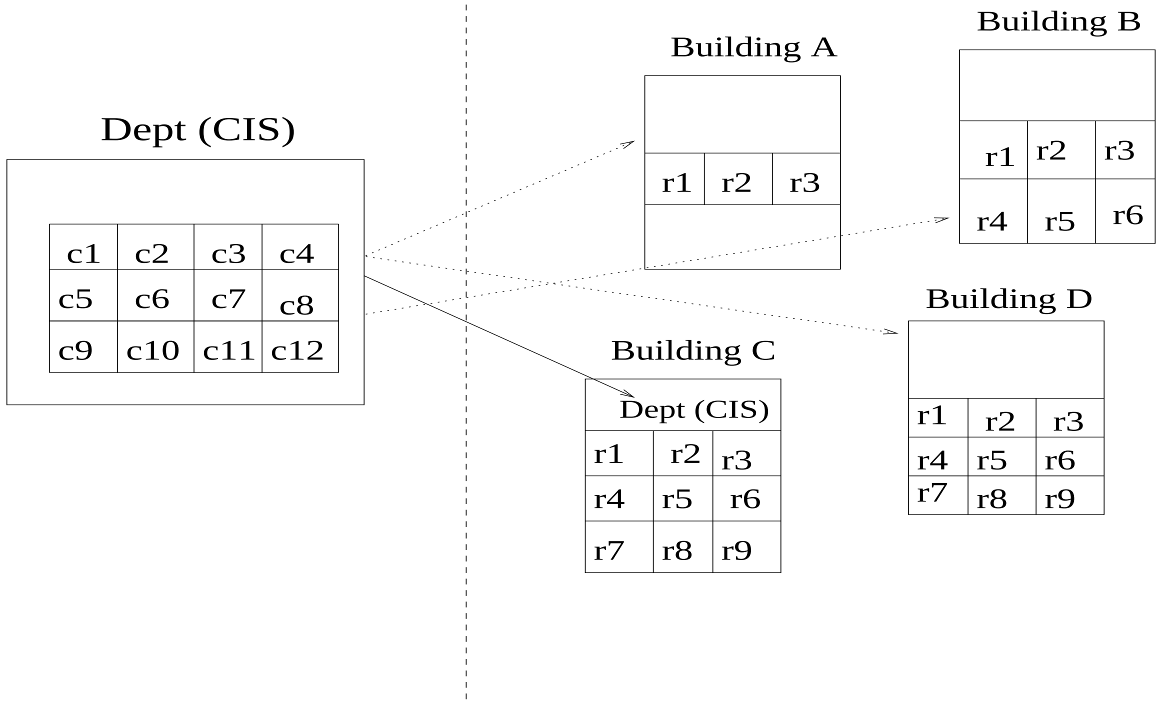

Figure 6.3: A schematic view of department(s) and their buildings. Here, it indicates that the home building for the CIS department is building C, and its classes can be scheduled in buildings A, B, and D, as well as C.

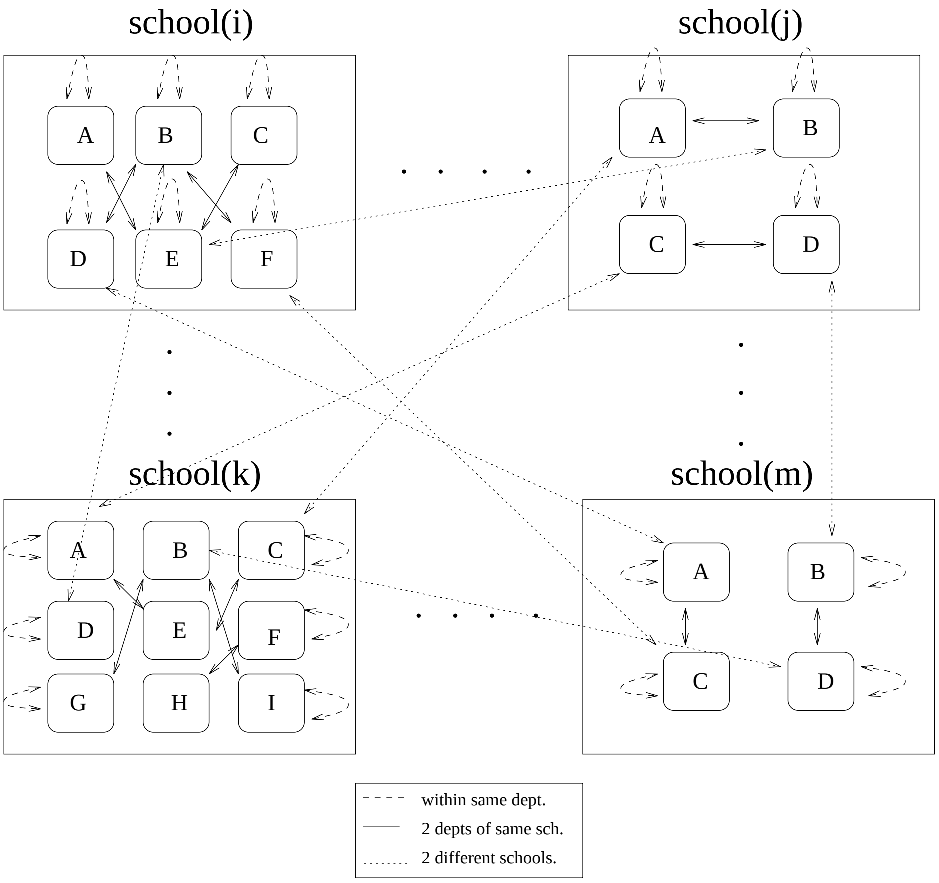

Figure 6.4: A schematic view of four different colleges (or schools) and their associated departments. The sketch shows class swapping between individual departments within the same school and others outside it. The dotted arrows denote swaps either within the same department or between two departments belonging to two different schools; while the sloid arrows denote swaps between departments within the same school.

Building Distances

Courses to Spread Out

Join-Listed Courses

Same-Day Courses

Simultaneous Courses

6.1.2 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, as illustrated in Figure 6.4.

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.

6.2 More Analysis of the Simulation

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 class scheduling 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. Johnson *et al.* [JM97] 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 6.1. 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 [GSP89, GSP92] consists of approximately 1000 students, 20 different possible majors, and an overall periodic school schedule (over weeks). In the case of Abramson *et al.* [Abr91], 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 6.1 lists all major components of the data we have used. Timetabling problems can be characterized by their *sparseness*. After the required number of classes 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 6.2 shows the sparseness of the three-semester data. Table 6.3 is also a three-semester sparseness ratio when the students are factored in, and closely correlated with that of Table 6.2. 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 soft constraints and thus are not necessarily satisfied in a valid solution.

Our overall results are shown in Tables 6.4 and 6.5. 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.

6.2.1 Experimental Results – Part I

...

...

6.2.2 Experimental Results – Part II

...

...

6.3 Conclusions

Academic course scheduling is only one example of a difficult planning problem; there are many similar problems occurring in the industry and the public sector. We have chosen this particular application since we feel that the problem is representative enough for this class of problems and also because real data were available to us.

Our successful approach is the application of simulated annealing to the the above-mentioned 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 [MEY95] which we also have used, 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 6.1: 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 6.2: 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 6.3: A three-semester sparseness ratio (including students) computed by $\mathcal{R}(N_{sp}/(N_x N_t)$, ((number majors * average number of classes taken per students) / number of students)). The smaller the ratio, the less sparse the problem is and for this problem the ratio is based on $N_t = 15$ time slots each with a duration of 55 minutes. The first slot starts at 8:00 am and the last slot ends at 9:45 pm.

Academic Time Period	sparseness ratio
First Semester	0.35
Second Semester	0.38
Third Semester	0.50

Table 6.4: 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 6.5: 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 7

Optimization Networks

As a prelude to mean field annealing (MFA), a summary of the Hopfield model is given below.

7.1 The Hopfield Network

7.1.1 Some Background

The network described by Hopfield in his seminal 1982 paper [Hop82] is in fact a special case of the *additive model* developed by Grossberg in the 1960's [Gro88]. Hopfield reviewed the network's application as a content addressable memory with both binary [Hop82] and continuous valued [Hop84] outputs. However, after detailed investigations into the network's performance by a number of authors, it is now clear that other types of content addressable memory are far more efficient [TTR91].

The Hopfield network was first proposed for combinatorial optimization applications in [HT85, HT86], where a penalty function mapping for the traveling salesman problem was given. The mapping was of a rather *ad hoc* nature, with separately weighted penalty functions for each of the hard constraints. The weights were set by trial and error, resulting, not surprisingly, in poor performance [WP88]. Nevertheless, some researchers preserved with this approach, sometimes proposing modified networks to correct what is essentially a sloppy mapping.

Real progress in the field can be identified with three main movements: rigorous investigation of network convergence, the development of more sensible mapping techniques, and the emergence of powerful annealing procedures.

- **Convergence**: Generalized feedback neural networks can exhibit oscillatory [CG83] or even chaotic [LG92] behavior. It is important to determine under what conditions a network will admit a Liapunov function for optimization applications. For simulated networks, convergence properties depend on whether the nodes are updated in serial or parallel mode: such issues are discussed (for discrete-valued networks) in [Bru90].

- **Mapping**: The development of rigorous mappings was delayed by the popularity of the original, *ad hoc* mapping of the TSP problem. Researchers concentrated on finding better weights for the penalty functions, often by way of an eigen vector analysis of Liapunov function [ANF90]: this, though correct, is quite complicated. Simpler, more reliable mappings began to appear for specific problems in [AF91]; some also dealt with inequality constraints [TCP88]. One particular rigorous mapping for the most general 0-1 quadratic programming problems onto standard Hopfield network dynamics was presented in [GP92]

- **Annealing**: In parallel with the development of rigorous mappings, modified dynamics were proposed as a way of improving the quality of any valid solutions found by the network. The most effective of these modifications typically represented some sort of annealing procedure. Mean field annealing was developed as an approximation to simulated annealing, and will be discussed from that viewpoint in the next section. More direct annealing procedures include hysteretc annealing [EDK$^+$91], convex relaxation [OB90], and matrix graduated non-convexity [AF91].

7.1.2 Structure and Dynamics of the Net

- The network is built by connecting a large number of neurons to each other to form a dense, parallel array. In general, any i^{th} neuron in the net is described by two variables: its current state (input) u_i, and its output v_i.

  The output is related to the input by a monotonically increasing transfer function: $v_i = g_\lambda(u_i)$.

- The net's dynamics are governed by the first order differential equation:

$$\frac{d\mathbf{u}}{dt} = -\eta\mathbf{u} + \mathbf{W}\mathbf{v} + \mathbf{I}^b \, . \tag{7.1}$$

- $g_\lambda(u_i) = \frac{1}{1+e^{-2\lambda u_i}}$, for large λ, this function approximates a step function, forcing v_i to $[0,1] \implies$ (the output of the net lies on the unit hypercube).

- W_{ij} is the connection strength between neuron i and neuron j.

- I_i^b is an input bias to neuron i.

- Neurons may be updated individually at random (asynchronously) or all together (synchronously).

7.1.3 Operation of the Network

The network (see Figure 7.1) operates as follows:

- The network's dynamics are governed by a set of nonlinear coupled differential equations of the form:

$$\frac{du_i}{dt} = -u_i + \sum_{j=1}^{n} W_{ij} v_j + I_i^b \, . \tag{7.2}$$

  or the general form of it:

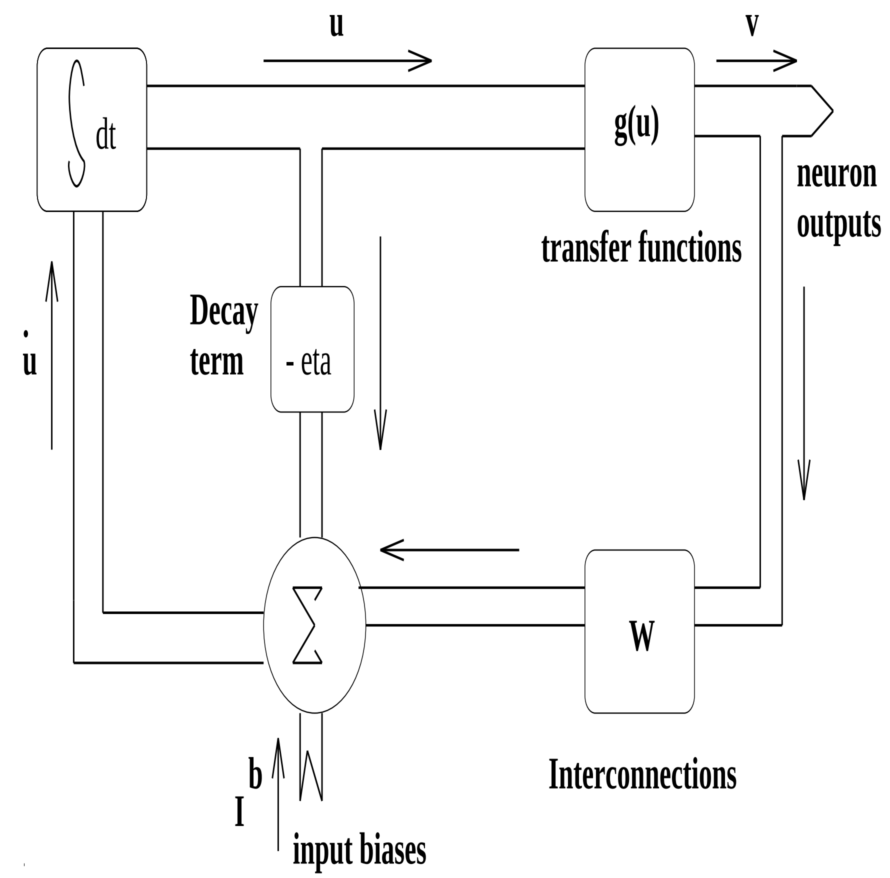

Figure 7.1: A schematic view of the Hopfield network.

$$\frac{d\mathbf{u}}{dt} = -\eta\mathbf{u} + \mathbf{W}\mathbf{v} + \mathbf{I^b} \ . \tag{7.3}$$

where $\mathbf{u}$, $\mathbf{W}$, $\mathbf{v}$, and $\mathbf{I}$ are vectors.

- The I_i^b corresponds to an input bias to neuron i.
- The η term introduces an element of decay into the network's dynamics, which can be used to ensure that $\mathbf{u}$ variables remain bounded in magnitude.

7.1.4 Updating of the States of the Neurons

To make sure that stability of the of the net is guaranteed, u_i is updated continuously by evaluating $\frac{d}{dt}u_i$ as a function of the total input to the i^{th} neuron.

$$\dot{\mathbf{u}} = -\eta\mathbf{u} + \mathbf{W}\mathbf{v} + \mathbf{I}^b \; , \tag{7.4}$$

where

$$v = g(u) \; . \tag{7.5}$$

Equations 7.4 and 7.5 govern the dynamics of the network.

7.1.5 Stability of the Net

Given that $\mathbf{W}$ is symmetric, Hopfield proved stability for the network by showing that with the above dynamic equations (7.4 and 7.5) the network output $\mathbf{v}$ evolves so as to minimize the *Liapunov function*:

$$E^{liap}(v) = \left(-\frac{1}{2}\sum_{ij} W_{ij}v_iv_j - \sum_i I_i^b v_i \right) + \eta \sum_i \int_0^{v_i} g^{-1}(V)\, dV \; . \tag{7.6}$$

- $v = (v_1 \ldots v_n)$ is denoting the net state, therefore we have an energy function defined throughout $v \in [0,1]^n$ and dynamics which guarantee convergence to minima.

- For a suitable W, the minima of the first term lie at the hypercube vertices. The second term is minimized at the hypercube center but has negligible impact at low η.

7.1.6 Updating the States of the Net

Updating of the states (or the output of the neurons) is carried out either asynchronously or synchronously. From a simulation point of view, we found that synchronous updating a bit more efficient, but from a theoretical point of view, it does not make much of a difference on the operation of the network as to which updating mode is used.

7.1.7 Mapping the Problem onto the Network

A representation of the problem is needed in which the feasible solutions lie at hypercube vertices. Each hypercube vertex is regarded as a configuration of the problem, with an associated energy (or cost) function. Next, the partition function is then formed by summing the Boltzmann factors of the admissible configurations. A quadratic energy function (to be mapped onto the first term of 7.6) for which the minima correspond to solutions and the depth of each minimum reflects the solution quality.

Hopfield and Tank used the TSP problem as an example, which they have mapped onto the net using a permutation matrix representation of the TSP tours.

7.1.8 Shortcomings of the Hopfield Model

The net's partition function sums over a vast number of configurations (e.g. for the TSP, all of the 2^{N^2} configurations are admissible). These configurations, again, for the TSP, are nothing like valid tours, and even though these offenders have small Boltzmann weights their large number inevitably affects the thermal average quantities.

7.2 Mean Field Annealing for Timetabling

7.2.1 Introduction and Motivation

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. Essentially, the *mean field approximation* involves averaging statistics of two coupled spins, and replacing a complex function with its truncated Taylor expansion around a *saddle point*. It is these approximations which differentiate mean field annealing from simulated annealing.

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, on which we would like to state few more points. We clearly know that the original Hopfield algorithm had difficulty constraining the network into valid solutions. Therefore, one possible way forward is to reduce the computational burden being placed on the net. For example, in the case of the TSP, this can be done by constraining each city to be "on" only once, by enforcing $\sum_a V_{ia} = 1 \ \forall i$, ($V_{ia} = 1/0$, which means that the city i is (is not) the a^{th} city visited in the tour), rather than relying on an energy penalty term to try to explicitly do this, as is the case in Hopfield model.

This idea was first utilized and analyzed for the TSP case by, among others, Peterson and Sӧderberg [PS89]. It greatly enhances the degree of convergence to valid solutions. Later, Peterson and colleagues somewhat extended the technique and applied it to a case of high school timetabling [GSP92]. In [PS89] the normalization (definition 7.2.1) occurs in the context of a *mean field annealing* approach. This generally yields the same final network equations as the "neuronal circuit" approach of Hopfield, but the exposition is a little bit clearer, as it is laid out in a statistical mechanics framework*. The idea is to regard each hypercube vertex as a configuration, with an associated energy (or cost) E. The partition function is then formed by summing the Boltzmann factors of the admissible configurations. For example, in the Hopfield algorithm, for the TSP, all of the 2^{N^2} configurations are admissible, whereas if each city is restricted to being visited only once, then only N^N vertices are admissible. After taking a mean field approximation, *saddle point equations* are derived (see next sections), the solutions of which pick out the dominant states of the

*This equivalence was also noted in Hopfield and Tank's 1985 paper [HT85]. Martin Simmen (in a personal communication) brought this point to our attention.

network at the current temperature T.

This statistical mechanics characterization gives a somewhat clearer picture of why the normalized model (which corresponds to a type of Potts model [Wu82] in physics, in that each city represented by a spin which can be in only one of N states) ought to perform better than Hopfield model. As indicated in [PS89], the partition function of Hopfield algorithm sums over a vast number of configurations which are nothing like valid tours in the TSP, and even though these offenders have small Boltzmann weights their large number inevitably affects the thermal average quantities. The other advantage that draws us to this derivation is that, by introducing a temperature, the concept of annealing, i.e. reducing T during a run, can be justifiably employed.

Th Mean-field annealing algorithm has been successfully applied to, as mentioned above, 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 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 Krogh, Hertz and Palmer [HKP91].

Definition 7.2.1 In [PS89], it was stated that the performance of MFA on certain problems can be improved using *neuron normalization*. This idea was originally inspired by an analogy with Potts glasses in statistical physics [KS87]. *Neuron normalization* modifies the MFA algorithm so that constraints of the form $\sum_{a \in \text{ States}} \nu_{ia} = 1$ are explicitly enforced, without the need for a penalty term. It was found by Peterson and other to be particularly useful for mapping the TSP and graph partitioning (GP) type problems; though the technique does not have general applicability. □

7.2.2 Brief Overview

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 smoothes 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:

$$U_{ia} = -\frac{1}{T}\frac{\partial E}{\partial V_{ia}} \tag{7.7}$$

$$V_{ia} = \frac{e^{U_{ia}}}{\sum_b e^{U_{ib}}}. \tag{7.8}$$

The MFA algorithm involves solving equations 7.7 and 7.8 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 7.8 around the trivial fixed-point [PS89, GSP92] $V_{ia}^{(0)} = \frac{1}{N_a}$, 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 7.7 and 7.8 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. [GSP92] The solutions correspond to stable states of the Hopfield model [HT85].

Observe from equation 7.8 that any solution to the MF equations respects a continuous version of the Potts condition

$$\sum_a V_{ia} = 1 \quad \forall\, i. \tag{7.9}$$

7.2.3 The Mean-Field Annealing Algorithm

The generic MFA algorithm appears in Figure 7.2. 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.* [GSP92] introduced a quantity called *saturation*, Σ, defined as

$$\Sigma = \frac{1}{N_i} \sum_{ia} V_{ia}^2 \ , \tag{7.10}$$

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, $\Sigma_{min} = \frac{1}{N_a}$

[†]Other measures could also do this, e.g. an entropy measure $-\sum_{ia} \nu_{ia} ln(\nu_{ia})$.

1. Choose a problem and encode the constraints into weights $\{w_{ij}\}$.
2. Find the approximate phase transition temperature by linearizing equation (7.8).
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 7.2: The Generic Mean-Field Annealing Algorithm

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.

7.2.4 Observations on the Algorithm

The first step of Figure 7.2 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 7.2) performs one update per neural variable (defined as one sweep) with sequential updating using equations 7.7 and 7.8. 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.* [GSP92].

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.

7.2.5 Definitions and Constraints

Here we define all of the terms and concepts used throughout this chapter for the implementation of class scheduling on the Potts model.

Let (N_p) to be a number of professors that give (N_q) classes in (N_x) rooms at (N_t) time slots. Event-wise, (p, q) event – denoting a professor p giving a class q – takes place in the space-time slot (x, t) – denoting room x at time t.

Definition 7.2.2 Let $S_{pq;xt}$ denote the Potts (or multi-state) neuron, where

$$S_{pq;xt} = \begin{cases} 1 & \text{if event } (p,q) \text{ takes place in } (x,t) \ , \\ 0 & \text{otherwise.} \end{cases} \tag{7.11}$$

□

Definition 7.2.3 We have dealt with two types of constraints:

1. Those involving professors, rooms, subjects, and time periods.

2. Those dealing with students scheduling and preferences.

In addition, some of these constraints are usually classified to be **hard**, others are **medium** and the rest are **soft**. □

Definition 7.2.4 Here, we define a number of **hard constraints** of the first type (see definition 7.2.3) .

- An event (p, q) should occupy precisely one space-time slot (x, t).

This constraint can be embedded in the neural net in terms of the Potts normalization condition:

$$\sum_{x,t} S_{pq,xt} = 1 \tag{7.12}$$

- Different events (p_1, q_1) and (p_2, q_2) should not occupy the same space-time slot (x, t).

$$E_{XT} = \frac{1}{2}\sum_{x,t}\sum_{p_1,q_1}\sum_{p_2,q_2} S_{p_1q_1;xt}S_{p_2q_2;xt} = \frac{1}{2}\sum_{x,t}\left(\sum_{pq} S_{pq;xt}\right)^2 \tag{7.13}$$

- A teacher p should have at most one class at a time.

$$E_{PT} = \frac{1}{2}\sum_{p,t}\sum_{q_1,x_1}\sum_{q_2,x_2} S_{pq_1;x_1t}S_{pq_2;x_2t} = \frac{1}{2}\sum_{p,t}\left(\sum_{qx} S_{pq;xt}\right)^2 \tag{7.14}$$

- A class q should have at most one teacher at a time.

$$E_{QT} = \frac{1}{2}\sum_{q,t}\sum_{p_1,x_1}\sum_{p_2,x_2} S_{p_1q;x_1t}S_{p_2q;x_2t} = \frac{1}{2}\sum_{q,t}\left(\sum_{px} S_{pq;xt}\right)^2 \tag{7.15}$$

$\square$

In [Hop84, HT85], Hopfield introduced a model with neurons having a graded response. So given I_i a fixed input bias to neuron x_i, W_{ij} the weight of the connection between neurons (x_i, x_j), and parameter T to denote the *Temperature*, the behavior of x_i can be stated as:

$$x_i = \frac{1}{2} + \frac{1}{2} \cdot tanh\left((\sum_{i \neq j} W_{ij}x_j - I_i) \times \frac{1}{T} \right) \tag{7.16}$$

When $T \longrightarrow 0$, function (7.16)'s response takes on a step-wise form. These models of [HT85] are known to be a bit sensitive to parameter tuning for real world applications, especially those of the type that involves "finding" or "searching for" 1 out-of N type problems.

Definition 7.2.5 At a finite T, x_i (of eq. 7.16) can be replaced by an MFT variable υ_{ia}, then differentiating to obtain the Potts MFT equations:

$$\upsilon_{ia} = -\frac{1}{T} \cdot \frac{\partial E[\vec{\nu}]}{\partial \nu_{ia}} \tag{7.17}$$

$$\nu_{ia} = \frac{e^{\upsilon_{ia}}}{\sum_b e^{\upsilon_{ib}}} \tag{7.18}$$

Which enforces

$$\sum_a \nu_{ia} = 1 \quad \forall i\,. \tag{7.19}$$

Where a denotes the vector components or all the elements of the group. Also, we could state that $\nu_{ia} \approx$ Probability(neuron(i) $\in$ state(a)). Equations (7.17, 7.18) are computed in an iterative manner. □

Definition 7.2.6 To carry out the minimization of the energy functions of definition (7.2.2) we need to convert the neural variables $\{0, 1\}$ of S_i to their corresponding *mean field variables* at T. These correspondences are derived (see [GSP92]) to be:

$$V_{pq;xt} \approx < S_{pq;xt} >_T \;; \tag{7.20}$$

which is the thermal average of the original binary spines, with the following multi-state MFT equations (see (7.17, 7.18) for a general format):

$$U_{pq;xt} = -\frac{1}{T} \cdot \frac{\partial E}{\partial V_{pq;xt}} \tag{7.21}$$

$$V_{pq;xt} = \frac{e^{U_{pq;xt}}}{\sum_{x't'} e^{U_{pq;x't'}}} \tag{7.22}$$

□

The MFT equations of υ and ν are solved *iteratively* either synchronously or asynchronously, under T. Also, these equations give rise to 2 phase transitions; one in x (space) and the other in t (time). This leads to factorization of the system by replacing $S_{pq;xt}$ by X-**neurons** $S^{(X)}_{pq;x}$ and $Time$-**neurons** $S^{(Time)}_{pq;t}$:

Definition 7.2.7

$$S_{pq;xt} = S^{(X)}_{pq;x} \cdot S^{(Time)}_{pq;t} \tag{7.23}$$

With separate Potts conditions:

$$\sum_x S^{(X)}_{pq;x} = 1 \tag{7.24}$$

$$\sum_t S^{(Time)}_{pq;t} = 1 \tag{7.25}$$

For X-**neurons** and $Time$-**neurons**, respectively. Rather than dealing with p and q directly, an independent variable (i) -acting as an event index- to which p and q are attributes $p(i)$ and $q(i)$. The value of these exits in a look-up table which contains all the relevant information to process each event (i). Now, the above (7.24, 7.25) conditions can be re-stated as follows:

$$\sum_x S^{(X)}_{i;x} = 1 \tag{7.26}$$

$$\sum_t S^{(Time)}_{i;t} = 1 \tag{7.27}$$

□

During annealing, as we go from a higher temperature to a lower one, a *phase transition* is passed at $(T = T_c)$, where T_c[‡] is the *critical temperature*. The value of T_c set the scale or the range of the value of T, and it is estimated by expanding 7.22 around some trivial fixed-point such as

$$V^{(0)}_{pq;xt} = \frac{1}{N_x N_t} \quad \text{, at } T = 0 \; . \tag{7.28}$$

Also T_c depends on the self-coupling coefficient β, and on the method of the update. This parameter β acts as a Lagrangian multiplier for the given energy constraint(s). See section on the generic Potts algorithm and [PS89].

[‡]See pp 29-31 of [HKP91] for an approximate value in the case of Ising ferromagnets.

Definition 7.2.8 *Relative Clamping*: During the revision process of an existing schedule, sometimes it is desirable to have a certain cluster of events to stick together in time (or within some specified period of time). This amounts to making the following notational replacement:

$$i \longrightarrow (j,k) \quad ; \tag{7.29}$$

where (j) denotes the cluster and (k) an event within (j). In this situation, one has a common t-neuron ($S_{j;t}^{(Time)}$) for cluster j but distinct x-neurons ($S_{jk;x}^{(X)}$) for the individual events. □

Definition 7.2.9 To deal with events that take place over two consecutive hours, what [PS89] called effective *time*-neurons $\tilde{S}_{i;t}^{(Time)}$ are introduced:

$$\tilde{S}_{i;t}^{(Time)} = \sum_{k=0}^{g_i-1} S_{i;t-k}^{(T)} \quad ; \tag{7.30}$$

where $g_i = 1$ for single hours and $= 2$ for double hours. For those events that occur between one and two hours, g_i can be adjusted accordingly to handle these cases. □

Utilizing definition (7.2.9), constraints (7.13, 7.14, 7.15) can be put into the following form:

Definition 7.2.10

$$E_{XT} = \frac{1}{2} \sum_{x,t} \sum_{i \neq i'} S_{i;x}^{(X)} S_{i';x}^{(X)} \tilde{S}_{i;t}^{(Time)} \tilde{S}_{i';t}^{(Time)} \tag{7.31}$$

$$E_{PT} = \frac{1}{2} \sum_{p,t} \sum_{i \neq i'} \delta_{p(i),p(i')} \tilde{S}_{i;t}^{(Time)} \tilde{S}_{i';t}^{(Time)} \tag{7.32}$$

$$E_{QT} = \frac{1}{2} \sum_{q,t} \sum_{i \neq i'} \delta_{q(i),q(i')} \tilde{S}_{i;t}^{(Time)} \tilde{S}_{i';t}^{(Time)} \tag{7.33}$$

□

Definition 7.2.11 The Kronecker $\delta_{q(i),q(i')}$ ensures that only events i and i' with identical "primordial" classes are, for example, summed over. Here, $q(i)$ and $q(i')$ refer to classes of the same type or 2 sections derived from the same original class.

$$\delta_{r,s} = \begin{cases} 1 & \text{if } r = s \\ 0 & \text{otherwise.} \end{cases} \tag{7.34}$$

□

Definition 7.2.12 One important soft constraint that need to be handled is *group formation*. Briefly, it can be stated as follows: for some optional subjects the classes are broken up into option groups temporarily forming new classes. In order to account for this, we need a formalism that allows for the breaking up of these primordial classes and subsequent recombinations into *quasi*-classes. This is a common practice with those classes where the students have many choices. To carry out such an extension to quasi-classes one might have contributing pairs of events where the primordial classes are different. Hence one should replace δ with a more structured overlap matrix Γ which is given by a look-up table.

$$\delta_{q(i),q(i')} \longrightarrow \Gamma_{q(i),q(i')} \ . \tag{7.35}$$

□

Definition 7.2.13 From definitions (7.2.8, 7.2.12), we need to make sure that $(\forall k)$ events within a given (j) cluster $p(j,k) \neq 0$. This says that there can be no "uncovered" class period or a class event (that is, periods when no professor has been assigned to an already scheduled class). For this purpose we introduce $\tilde{\delta}_{j,k}$ according to:

$$\tilde{\delta}_{j,k} = \delta_{j,k}(1 - \delta_{j,0}\delta_{k,0}) \ . \tag{7.36}$$

If $j = 0$ then $\forall k \ , \ k = 0$. □

Definition 7.2.14 Utilizing the concepts of the above definition (7.2.12, 7.2.13), the collision terms of definition (7.2.10) are extended to ensure that all the hard constraints are satisfied (or the solutions obtained are legal). The extension takes the following forms:

$$E_{XT} \text{ (form 1) } = \frac{1}{2} \sum_{jk \neq j'k'} \sum_{x} S^{(X)}_{jk;x} S^{(X)}_{j'k';x} \sum_{t} \tilde{S}^{(Time)}_{j;t} \tilde{S}^{(Time)}_{j';t} \tag{7.37}$$

$$E_{XT} \text{ (form 2) } = \frac{1}{2} \sum_{j \neq j'} \sum_{kk'} \sum_{x} S^{(X)}_{jk;x} S^{(X)}_{j'k';x} \sum_{t} \tilde{S}^{(Time)}_{j;t} \tilde{S}^{(Time)}_{j';t} + \frac{1}{2} \sum_{j} g_j \sum_{k \neq k'} \sum_{x} S^{(X)}_{jk;x} S^{(X)}_{jk';x} \tag{7.38}$$

$$E_{PT} \text{ (form 1) } = \frac{1}{2} \sum_{p,t} \left[\sum_{jk} \tilde{\delta}_{p,p(j,k)} \tilde{S}^{(Time)}_{j;t} \right] \left[\sum_{j' \neq j} \sum_{k'} \tilde{\delta}_{p,p(j',k')} \tilde{S}^{(Time)}_{j';t} \right] \tag{7.39}$$

$$E_{PT} \text{ (form 2) } = \frac{1}{2} \sum_{j \neq j'} \sum_{kk'} \tilde{\delta}_{p(j,k),p(j',k')} \sum_{t} \tilde{S}^{(Time)}_{j;t} \tilde{S}^{(Time)}_{j';t} \tag{7.40}$$

$$E_{QT} \text{ (form 1) } = \frac{1}{2} \sum_{qt} \left[\sum_{jk} \delta_{q,q(j,k)} \tilde{S}^{(Time)}_{j;t} \right] \left[\sum_{j' \neq j} \sum_{k'} \Gamma_{q,q(j',k')} \tilde{S}^{(Time)}_{j';t} \right] \tag{7.41}$$

$$E_{QT} \text{ (form 2) } = \frac{1}{2} \sum_{j \neq j'} \sum_{kk'} \Gamma_{q(j,k),q(j',k')} \sum_{t} \tilde{S}^{(Time)}_{j;t} \tilde{S}^{(Time)}_{j';t} \tag{7.42}$$

Note: Each "form 2" above is a derivation from its corresponding "form 1". □

The factorization idea of (dfn 7.2.14) can be extended to *mean field* variables $V^{(X)}_{jk;x}$ and $V^{(Time)}_{j;t}$, corresponding to neurons ($S^{(X)}_{jk;x}$) and ($S^{(Time)}_{j;t}$), respectively. $V^{(X)}_{jk;x}$ and $V^{(Time)}_{j;t}$ can be interpreted as being the probabilities that the corresponding events occur at given x-value and t-value with normalization given by the above equations (7.26, 7.27) .

Substituting V for S in (7.26, 7.27), the *mean field* equations at a finite T, are given in terms of the *local fields* $U^{(X)}_{jk;x}$ and $U^{(Time)}_{j;t}$. Now, the *mean field* dynamics proceed over the following local and mean field variables:

Definition 7.2.15

$$U_{jk;x}^{(X)} = -\frac{1}{T} \cdot \frac{\partial E}{\partial V_{jk;x}^{(X)}} \tag{7.43}$$

$$U_{j;x}^{(Time)} = -\frac{1}{T} \cdot \frac{\partial E}{\partial V_{j;t}^{(Time)}} \tag{7.44}$$

$$V_{jk;x}^{(X)} = \frac{e^{U_{jk;x}^{(X)}}}{\sum_{x'} e^{U_{jk;x'}^{(X)}}} \tag{7.45}$$

$$V_{j;t}^{(Time)} = \frac{e^{U_{j;t}^{(Time)}}}{\sum_{t'} e^{U_{j;t'}^{(Time)}}} \tag{7.46}$$

The iterations of the *MF* dynamics over equations (7.43 - 7.46) can either be **synchronous** or **serial** (see [GSP92]). □

Definition 7.2.16 *Synchronous Updating*: A single sweep consists of first computing all the local fields, and then updating all the neurons. That means do equations (7.43,7.44) then perform equations (7.45,7.46). □

Definition 7.2.17 *Serial Updating*: For each neuron $S_{jk;x}^{(X)}$ and/or $S_{j;t}^{(Time)}$, the *local field* is computed immediately before the corresponding *neuron state* is updated. That means either do equation (7.43) followed by (7.45) and/or do equation (7.44) followed by (7.46). □

Next, we need to define 2 important **soft constraints** encountered when scheduling classes.

Definition 7.2.18

- *Spreading*: different classes in a particular area (i.e. computer science) should be spread over the week in a (MWF) and (TuTh) type of spread.

First, assume we have access to a subject attribute $s(j,k)$. Second, we define an effective "day-neuron" as follows ($\tilde{S}^{(Time)}_{j;dh}$ is reduced to a D-type neuron by summing over all hours of the day):

$$S^{(D)}_{i;d} = \sum_h \tilde{S}^{(Time)}_{j;dh} = g_j \cdot S^{(Time)}_{j;dh} \tag{7.47}$$

So the penalty term E_{QSD} that spreads classes Q in a particular subject S over different week-days D :

$$E_{QSD} = \frac{1}{2}\sum_{qsd}\left[\sum_{jk}\sum_h \delta_{q,q(j,k)}\delta_{s,s(j,k)}\tilde{S}^{(Time)}_{jk;dh}\right]^2 \tag{7.48}$$

Refining the above a bit more by considering 2 different clusters (j) and $(\hat{j})$ of classes over D : (q and s are not used in the summation, since no change occurs to them)

$$E_{QSD} = \frac{1}{2}\sum_d\sum_{j\neq\hat{j}}\sum_{kh}\sum_{\hat{k}\hat{h}} \delta_{q(j,k),q(\hat{j},\hat{k})}\delta_{s(j,k),s(\hat{j},\hat{k})}\hat{S}^{(Time)}_{j;dh}\hat{S}^{(Time)}_{\hat{j};d\hat{h}} \tag{7.49}$$

Now for some day (d) of the week:

$$E_{QSD} = \frac{1}{2}\sum_{j\neq\hat{j}}\sum_{k\hat{k}} \delta_{q(j,k),q(\hat{j},\hat{k})}\delta_{s(j,k),s(\hat{j},\hat{k})}\sum_d S^{(D)}_{j,d}S^{(D)}_{\hat{j},d} \tag{7.50}$$

- *Glueing*: The schedule should have as few "holes" as possible; that is lessons should be glued together.

The penalty term that rewards situations where (d,h)-events are glued to $(d,h-1)$-events can be defined as follows:

$$E_{QDH} = -\alpha\sum_q\sum_{d,h}\left[\sum_{j,k}\delta_{q,q(j,k)}\tilde{S}^{(Time)}_{j;dh}\right]\left[\sum_{\hat{j}\neq j}\Gamma_{q,q(\hat{j},\hat{k})}\tilde{S}^{(Time)}_{j;d(h-1)}\right] \tag{7.51}$$

The energy term of (7.51) can be summarized to say that: *if cluster (j) with events each symbolized by (k) occurs on hour (h) of day (d) then it would be quite desirable* *to have another event ($\hat{k}$) from a different cluster ($\hat{j}$) to occur on hour ($h-1$) of that day.*

Next, since (q) is fixed we could re-write the above equation as follows:

$$E_{QDH} = -\alpha \sum_{\hat{j}\neq j} \sum_{k,\hat{k}} \Gamma_{q(j,k),q(\hat{j},\hat{k})} \sum_{d,h>1} \tilde{S}^{(Time)}_{j;dh} \tilde{S}^{(Time)}_{\hat{j},d(h-1)} \tag{7.52}$$

The presence of the ($-$) sign in front of (7.51, 7.52) indicates that such energy functions are to be minimized. Also, the (α) in front of the the above forms governs the strength of this reward relative to the energies of equations (7.13,7.14,7.15). This (α) takes on a value between zero and 1.

Finding *legal* solutions with no collisions corresponds to minimizing the *hard* energy:

$$E_{hard} = combine(E_{XT}, E_{PT}, E_{QT}) \tag{7.53}$$

to zero. The *soft energy* is given by:

$$E_{soft} = combine(E_{QSD}, E_{QDH}) \ . \tag{7.54}$$

The total energy E to be minimized is then:

$$E = combine(E_{hard}, E_{medium}, E_{soft}) \ . \tag{7.55}$$

□

7.2.6 More Definitions and Constraints

Let $q(i)$ represents a section of class q, $O_{q(i)}$ indicates that section $q(i)$ is open, and $C_{q(i)}$ is closed. Also, let $b(x,t)$ be a balanced interval for some time period (t), R is the overall set of students, and an arbitrary given subset is $r \subseteq R$.

Here, we need to introduce the R-neuron: $S^{(R)}_{r(i);(d(t)d(x)}$ where $k = 1, 2, \ldots$. Each student $r(i) \in R$ should have a balanced schedule over the two-group of days (MWF) and ($TuTh$). We will also use r_s to denote a single student.

For X and T neurons, let $x(j)$ and $t(j)$ where $j = 1, 2, \ldots, n$ stand for places and time slots (or events), respectively. Using such notation, an example of the Potts normalization will be: $S^{(X)}_{p(i)q(i);x(j)}$ and $S^{(Time)}_{p(i)q(i);t(j)}$ respectively, where $i = 1, 2, \ldots, n$ is a section index.

Definition 7.2.19 *The athletes case*: Let $\vec{r} \subseteq R$ denotes the group of athletes that are dealt with in this study. Let (r^*,b) be playing/training events scheduled during the weekdays at (or during) times when academic (attending lectures) events (r^+,a) are also scheduled, or may possibly occur. Let ($d(x)$, $d(t)$) a place/time during the week at which any of these events can occur.

In addition to the above stated Potts neurons, we also will be using :

1. $S^{(\vec{r},X)}_{r^*b;d(x)}$, $S^{(\vec{r},Time)}_{r^*b;d(t)}$: It takes a value of (1) if event (r^*, b) takes place in the place and time slot, respectively, of a weekday d, and zero otherwise.

2. $S^{(\vec{r},X)}_{r^+a;d(x)}$, $S^{(\vec{r},Time)}_{r^+a;d(t)}$: It takes a value of (1) if event (r^+, a) takes place in the place and time slot, respectively, of a weekday d, and zero otherwise.

3. $S^{(\vec{r},X)}_{jk;d(x)}$: It takes a value of (1) if event (k) of cluster (j) that is associated with the x-neuron of $\vec{r}$ occurs in place $d(x)$, and zero otherwise.

4. $S^{(\vec{r},X)}_{\hat{j}\hat{k};d(x)}$: It takes a value of (1) if event $(\hat{k})$ of cluster $(\hat{j})$ that is associated with the x-neuron of $\vec{r}$ occurs in place $d(x)$, and zero otherwise.

5. $\tilde{S}^{(\vec{r},Time)}_{j;d(t)}$: It takes a value of (1) if cluster (j) that is associated with the t-neuron of $\vec{r}$ occurs in time $d(t)$, and zero otherwise.

6. $\tilde{S}^{(\vec{r},Time)}_{\hat{j};d(t)}$: It takes a value of (1) if cluster $(\hat{j})$ that is associated with the t-neuron of $\vec{r}$ occurs in time $d(t)$, and zero otherwise.

In the above list, the last 2 neurons act as what Peterson calls *effective time-neurons*. Not only will they handle periods of single hours but also variants of longer and shorter intervals. □

Definition 7.2.20 Set (ϵ) and (c) to some desired values depending on the time period of the set of classes, now, the above *effective* t-neurons can be defined as follows:

$$\tilde{S}^{(\vec{r},T)}_{j;d(t)} = \sum_{k=0}^{g_j - c} S^{(\vec{r},T)}_{j;d(t-k+\epsilon)} \tag{7.56}$$

$\tilde{S}^{(\vec{r},Time)}_{\hat{j};d(t)}$ can be defined in a similar manner. If $(c = 1)$ and $(g_i = 2)$, then we are dealing with double-hour intervals so we set $(\epsilon = 0)$. If $(c = 1)$ and $(g_i = 1)$ then $(\epsilon = 0)$ and

$$\tilde{S}^{(\vec{r},Time)}_{j;d(t)} = S^{(\vec{r},Time)}_{j;d(t)}. \tag{7.57}$$

□

Definition 7.2.21 Let $S^{(R)}_{q(i)r(i);b(x,t)}$ for balancing the enrollment in a multi-section courses. Section $q(i)$ of students $r(i)$ are balanced over space-time $b(x,t)$.

To enroll student $r(i)$ in a particular section $q(i)$ given the tag EC to indicate the status of the student, we have $S^{(X[Time])}_{q(i)r(i);(d(x)[d(t)])}|EC \in \{0,1\}$ is set to0 if EC is 0, otherwise, it is set to 1 and the action $q(i)r(i)$ occurs during place $d(x)$ and/or time $d(t)$.

Let $lin(c_i, c_j)$ representing the concept that course c_i is linked to some other course c_j ; in this case, c_i is a pre-requisite to c_j. The degree of "linkage" varies between 0 and 1. It takes a value of 1 if under all circumstances, course c_i must be taken before c_j.

To deal with this link concept, we assumed in definition (7.2.18) that we have an attribute $s(j,k)$ to represent a subject. For a strong linkage, we will assume that we're dealing with subjects that belong to the same cluster (j) (e.g. physics courses), so the link between various events of (j) is: $lin(s(j,k), s(j,\hat{k}))$.

Other degrees of linkage can involve subjects or courses from different groups or clusters (e.g. physics and math courses, or engineering and math courses), and this connection is in the form: $lin(s(j,k), s(\hat{j},\hat{k}))$.

Let $\hat{x}(i)$ be a single location (or a seat) in class $x(i)$ such that $i = 1, 2, \ldots, num$; where (num) is the number of class-rooms. □

Day-Class Balancing

Each student of any particular section (i.e. $r(k)$) should have a balanced schedule between (*MWF*) and (*TuTh*). The balancing operation is accomplished by spreading as much as possible the courses of the student over each of (*MWF*) and (*TuTh*).

Let D be the set $\{\{MWF\},\{TuTh\}\} = \{D_1, D_2\}$. Assume the existence of the subject attribute $s(j,k)$. We introduce the energy function E_{QSD} that spreads the classes (Q) in a particular field or subject (S) over the set $\{D_1, D_2\}$.

Definition 7.2.22 The Potts condition of balancing $S^{(\mathbf{R})}_{q(m)r(m);b(x,t)}$ (take a value of 1 for balanced and 0, otherwise) is accomplished through one (or all) of these energies:

$$E^r_{QSD} = \frac{1}{2}\sum_{j\neq\hat{j}}\sum_{k\hat{k}} \delta_{q(j,k),q(\hat{j},\hat{k})}\delta_{s(j,k),s(\hat{j},\hat{k})} \sum_{d_1,d_2} S^{(D)}_{j,d_1} S^{(D)}_{\hat{j},d_2} \tag{7.58}$$

$$E^r_{QSD_1} = \frac{1}{2}\sum_{j_1\neq\hat{j}_1}\sum_{k_1\hat{k}_1} \delta_{q(j_1,k_1),q(\hat{j}_1,\hat{k}_1)}\delta_{s(j_1,k_1),s(\hat{j}_1,\hat{k}_1)} \sum_{d_1}\sum_{i\neq\hat{i}} S^{(D)}_{i,d_1} S^{(D)}_{\hat{i},d_1} \tag{7.59}$$

$$E^r_{QSD_2} = \frac{1}{2}\sum_{j_2\neq\hat{j}_2}\sum_{k_2\hat{k}_2} \delta_{q(j_2,k_2),q(\hat{j}_2,\hat{k}_2)}\delta_{s(j_2,k_2),s(\hat{j}_2,\hat{k}_2)} \sum_{d_2}\sum_{i\neq\hat{i}} S^{(D)}_{i,d_2} S^{(D)}_{\hat{i},d_2} \tag{7.60}$$

□

The Case of Student Athletes

The main constraint of this case very much analogous to the **hard** constraint stated somewhere above : *different events should not occupy the same space-time slot.*

Definition 7.2.23

$$E^{\vec{r}}_{d(XT)} = \frac{1}{2}\sum_{d}\sum_{xt}\sum_{\dot{r}b}\sum_{\ddot{r}a} S^{(\vec{r},X[Time])}_{\dot{r}b;d(x)[d(t)]} S^{(\vec{r},X[Time])}_{\ddot{r}a;d(x)[d(t)]} \tag{7.61}$$

Where $(\dot{r}b)$ and $(\ddot{r}a)$ are some two different events. After some further refinement, we obtain:

$$E^{\vec{r}}_{d(XT)} = \frac{1}{2}\sum_{j\neq\hat{j}}\sum_{k\hat{k}}\sum_{d(x)} S^{(\vec{r},X)}_{jk;d(x)} S^{(\vec{r},X)}_{\hat{j}\hat{k};d(x)} \sum_{d(t)} \tilde{S}^{(\vec{r},Time)}_{j;d(t)} \tilde{S}^{(\vec{r},Time)}_{\hat{j};d(t)} + \frac{1}{2}\sum_{j} g_j \sum_{k\neq\hat{k}}\sum_{d(t)} S^{(\vec{r},X)}_{jk;x} S^{(\vec{r},X)}_{j\hat{k};x} \tag{7.62}$$

Since it is always the case that athletic training facilities are kept separate from the academic class-rooms and lecture halls, a more refined energy function is obtained:

$$E^{\vec{r}}_{d(Time)} = \frac{1}{2}\sum_{j\neq\hat{j}}\sum_{d(t)} \tilde{S}^{(\vec{r},Time)}_{j;d(t)} \tilde{S}^{(\vec{r},Time)}_{\hat{j};d(t)} \tag{7.63}$$

□

Eligibility for Enrollment

Definition 7.2.24 Before going through the process of balancing, each student ($r(i) = r_s$) eligibility for enrolling in a given class need to be determined. Let the student's eligibility criteria be denoted by (EC), using the notation stated above, we could derive the following energy function —note: if EC is set to "No" then it takes a 0 value in the energy function, otherwise, it is a 1 :

$$E_{QR} = \frac{1}{2}\sum_{qr}\sum_{jk}\sum_{i} \delta_{q,q(j,k)}\delta_{r,r(i)} S^{X}_{ij;x(i)} |(EC \in \{0, 1\}) \tag{7.64}$$

$$\sum_{i} O_{q(i)} = 1 \Rightarrow \text{ Test: } \{EC(R) + \text{ satisfy-constraints } (Hard, Soft)\} \tag{7.65}$$

□

Course Pre-requisite and Subsequent Assignment

Utilizing the link concept ($\mathbf{lin(s, s')}$) between two different subjects (see definition 7.2.21), a general energy assignment can be put together as follow:

Definition 7.2.25 Let (i) stands for the situation at hand (also can be viewed as a space-time event).

$$E_{link(1)} = -\gamma \sum_{is} \left[\sum_{jk} \delta_{s,s(j,k)} \tilde{S}^{(X[Time])}_{ln(s,s');i} \right]^2 \tag{7.66}$$

$$E_{link(2)} = -\gamma \sum_{s} \sum_{j \neq j'} \sum_{k \neq k'} \delta_{s(j,k),s(j',k')} \sum_{i} \tilde{S}^{(X[Time])}_{ln(s(j,k),s(j',k'));i} \tilde{S}^{(X[Time])}_{ln(s(j,k),s(j,k'));i} \tag{7.67}$$

$$E_{link(3)} = -\gamma \sum_{j \neq j'} \sum_{k \neq k'} \delta_{s(j,k),s(j',k')} \tilde{S}^{(X[Time])}_{ln(s(j,k),s(j',k'));i} \tilde{S}^{(X[Time])}_{ln(s(j,k),s(j,k'));i} \tag{7.68}$$

If $(\gamma > 0)$ then function minimization is the target here, otherwise maximization will take place.

□

Enroll in all parts of a course

There are a number course of different types that not only have lecture periods but also associated sections that consist of either labs or recitations or both. Signing up for such a course entails enrolling in all associated sections of it.

Let us assume the existence of some subject attribute $s(j,k)$ consisting of 2 components: $\dot{s}$ denoting lectures, and $\ddot{s}$ stands for labs. Now the energy equation depicting this assignment can be outlined as follow:

$$E_{enroll(1)} = \frac{1}{2} \sum_{s} \sum_{jk} \sum_{i} \delta_{s,s(j,k)} \delta_{r,r(i)} \sum_{t} S^{(Time)}_{jk;t} \tag{7.69}$$

$$E_{enroll(2)} = \frac{1}{2} \sum_{si} \sum_{t \neq t'} \left[\sum_{jk} \delta_{s,\dot{s}(j,k)} \delta_{r,r(i)} S^{(Time)}_{jk;t} \right] \left[\sum_{jk'} \delta_{s,\ddot{s}(j,k')} \delta_{r,r(i)} S^{(Time)}_{jk';t'} \right] \tag{7.70}$$

$$E_{enroll(3)} = \frac{1}{2} \sum_{k \neq k'} \delta_{s,\dot{s}(j,k)} \delta_{s,\ddot{s}(j,k')} \sum_{t \neq t'} S^{(Time)}_{jk;t} S^{(T)}_{jk';t'} \tag{7.71}$$

Lecture hall capacity

In order to satisfy this constraint, the following inequality need to hold.

$$\forall x \sum_i ||x(i)| - \sum_j \sum_k S^{(X)}_{jk;x(i)}| \geq 0 \; . \tag{7.72}$$

Student status and priority mapping

$$E_{assign} = \frac{1}{2} \sum_{m,\hat{x},x} \sum_{i \neq i'} S^{(X)}_{\hat{x};x} \left(S^{(R)}_{r(i);(\hat{x}(m) \in x(m))} |(M = Req) + S^{(R)}_{r(i');(\hat{x}(m) \in x(m))} |(M = \chi) \right) \tag{7.73}$$

On the Energy Equation of the Potts Model

The energy function of the Potts spin model is of the form:

$$E = E_A + E_B + E_C + \ldots \tag{7.74}$$

In [PS89], in which the TSP is used as study case, parameters α and β are used to control the strength of E's restraints. The same parameters can also be used when dealing with non-Euclidean type CO problems such as scheduling.

Observation 7.2.26 For timetabling, β is split into two parts ($\beta^{(Time)}$ and $\beta^{(X)}$), one with respect to the time-based neurons and the other with respect to the space-based neurons. E_β becomes as follows:

$$-\frac{\beta^{(Time)}}{2} \sum_j \sum_t [S^{(T)}_{j;t}]^2 - \frac{\beta^{(X)}}{2} \sum_{jk} \sum_x [S^{(X)}_{jk;x}]^2 \tag{7.75}$$

Here, the appearance of $\beta^{(Time)}$ and $\beta^{(X)}$ is only in the diagonal terms. They play a crucial role in monitoring the dynamics (phase transitions, etc.). These diagonal terms correspond to self coupling interactions for the neurons of the network. In our simulations, each is set to -0.1. □

7.2.7 General Comments on the Algorithm

Let (N) denotes the total number of the utilized neurons. At high T, the mean field solutions will tend to be states near to the (symmetrical) maximum entropy state: $\forall(i,a)v_{ia} = 1/N$. Conversely at low T, finding a mean field solution will be equivalent to using a local optimization method on the internal energy — a procedure highly sensitive to the initial conditions and known to be ineffective (see p. 149 of [HT85]).

7.2.8 Some Implementation Details

The general idea is to initialize the system close to the symmetric state at a temperature near T_c, then, allowing the system to converge to a fixed point at each temperature, anneal (i.e. reduce T) until either the system reaches a stable high saturation state or some other termination criterion is satisfied. Specifically, the implementation used here has the following ingredients[§]: (Martin Simmen (in a private communication) investigated in depth these and most of the other issues pertaining to the Potts model, using the TSP as a test case.)

- **Initial State** Another possible initial value of $\nu_{ia}^{n=0}$ is $(1+\xi_{ia})/N$, with the ξ_{ia} being random variables drawn uniformly from $[-\xi,\xi]$. Clearly ξ should be $\ll 1$.

- **Convergence** At each T the ν_{ia} variables are updated as above or by the following rule[¶]:

$$\nu_{ia}^{(n+1)} = \nu_{ia}^{(n)} + \gamma(f_{ia} - \nu_{ia}^{(n)}), \quad 0 < \gamma \leq 1 \; . \tag{7.76}$$

  Setting $\gamma = 1$ recovers the dynamics of [PS89], and the $\gamma \to 0$ limit gives the first order Euler integration method for the differential equation:

$$d\nu_{ia}/dt = f_{ia} - \nu_{ia} \; . \tag{7.77}$$

[§]These follow the method used in [PS89] but with some additions/improvements done by the authors -see algorithm above.

[¶]Thanks to M. Simmen for pointing this out. Also see [PS89]

The fixed points (also referred to as the free energy minima) of this equation constitute the mean field solutions. The degree of convergence is monitored through some sort of a "tolerance" quantity of the form $\sum_{ia} (\nu_{ia} - f_{ia})^2$.

- **Annealing Schedule** A simple exponential scheme is used, i.e. after reaching a fixed point at T, $T \mapsto T \cdot T_r$, where $T_r \in (0, 1)$ controls the cooling rate. Roughly speaking, T_r should be set to 0.9.

- **Termination** One criterion is as outlined above in the algorithm. The other two involve terminating runs which either fail to reach a fixed point within a certain number of sweeps at some temperature, or which allow T to fall below some T_{min}, ($T_{min} \ll T_c$), without ever satisfying the $\Sigma > \Sigma^{thresh}$ criterion. T_{min} was set to 10^{-3}.

7.2.9 Performance Issues of MFA on Timetabling

Given M_x = average number of rooms for each category of classes, N_x = the overall total number of rooms, and N_t = total number of slots , putting the figures of the students aside, with a computational complexity of $O(M_x N_x N_t)$ per sweep, software implementation of the MFA are unlikely to be competitive with conventional serial algorithms such as 3-opt and Lin-Kernighan's, which empirically have an overall $O(N^2)$ running time [JM97]. Ideally, one would like to study the likely performance of the method through numerical simulations using a small step-size γ to approximate continuous time dynamics. However, this is computationally very expensive: not only do the simulation times grow as $\frac{1}{\gamma}$, but also small values of (γ) accentuate the difference between the expected value T_c^{exp} of the critical temperature T_c and the theoretically computed value T_c^{theo}. That is, low γ runs tend to depress T_c^{exp} artificially and thereby degrade solution quality, and as we observed, result in a number of constraint violations. Many of these constraints are hard, but the majority are classified to be soft. This later point can be compensated for by more stringent annealing parameters, but at the cost of longer simulation times. No attempts have been made to carry this out.

Also, we observe that the overall quality of the solution degrades as the problem size increases. This degradation moves from soft constraint violation to major violations of hard constraints, and hence, illegal timetables.

7.2.10 Mapping Timetabling onto the Potts Neural Nets

The Relationship Between the Key Timetabling Entities

Our problem consist of the following sets: teachers (P), classes (C), students (S), rooms (R), and time periods ($\mathcal{I}$).

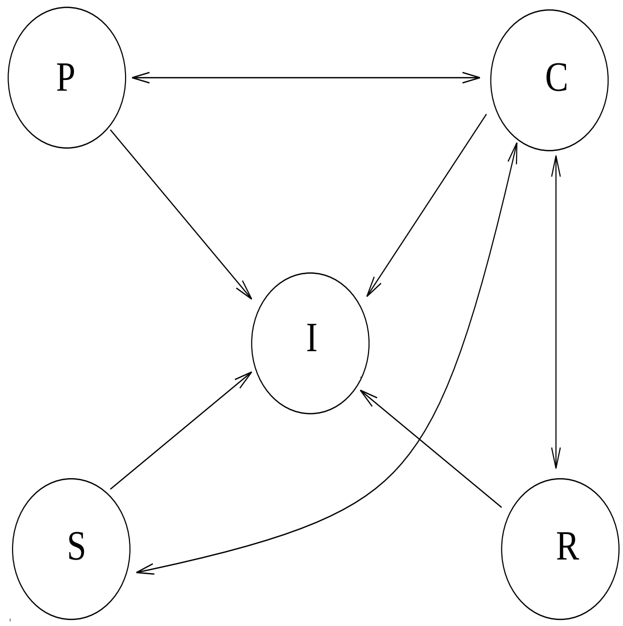

Figure 7.3: A one-way arrow indicates that the "bottom" entity depends on the "top" one. A two-way arrow indicates a mutual dependency between the two entities.

Neural Mapping

Involve mapping the problem onto the Potts model.

- These multi-state neurons, forming the network, represent different activities of the timetabling problem.
- Construct the constraints of the problem using these neurons.
- Derive those MF equations controlling the dynamics of the net:
  - Introduce the MF variables , for example,

$$V_{pq;xt} =< S_{pq;xt} >_T \quad . \tag{7.78}$$

  - Start off with the partition function:

$$Z = \sum_{[\vec{s}]} e^{-E[\vec{s}]/T} \ . \tag{7.79}$$

  - Re-write Z in terms of a multidimensional integral over continuous vectors $(\vec{u}_i, \vec{v}_i)$.
  - The MFT approximation to $< \vec{s}_i >$ is given by the value $\vec{u}_i$ at the saddle point equations:

$$\frac{\partial E}{\partial v_{ia}} = 0 \tag{7.80}$$

and

$$\frac{\partial E}{\partial u_{ia}} = 0 \ , \tag{7.81}$$

generating a set of self-consistency equations in the MF variables:

$$u_{ia} = -\frac{1}{T}\frac{\partial E[\vec{v}]}{\partial v_{ia}} \ , \tag{7.82}$$

and

$$v_{ia} = \frac{e^{u_{ia}}}{\sum_b e^{u_{ib}}} \ . \tag{7.83}$$

Where a denotes the vector components.

For events (pq, xt), we get the following MF variables:

$$u_{pq,xt} = -\frac{1}{T}\frac{\partial E}{\partial v_{pq,xt}} , \tag{7.84}$$

and

$$v_{pq,xt} = \frac{e^{u_{pq,xt}}}{\sum\limits_{x't'} e^{u_{pq,x't'}}} . \tag{7.85}$$

From the later follows, that $\sum\limits_{a} v_{ia} = 1$. One can think of the mean field v_{ia} as the probability for the Potts neuron i to be in state a.

- Iterate $v_{pq,xt}$ equation with annealing, i.e. starting at a high temperature, and successively lowering it in the course of the process. A phase transition is passed at $T = T_c$.

- Estimate the critical temperature T_c, which sets the scale of T, by expanding the above $v_{pq,xt}$ equation around the trivial fix-point : at $T = 0$

$$v^{(0)}_{pq,xt} = \frac{1}{N_x N_t} . \tag{7.86}$$

Chapter 8

Scheduling as a Graph Coloring Problem

8.1 introduction

Many scheduling problems involve allowing for a number of pairwise restrictions on which jobs can be done simultaneously. For instance, in attempting to schedule classes at a university, two courses taught by the same faculty member cannot be scheduled for the same time slot. Similarly, two courses that are required by the same group of students also should not conflict. The problem of determining the minimum number of time slots needed subject to these restrictions is a graph coloring problem (GC).

GC has considerable application to a large variety of complex problems involving optimization. Among those problems is the one we have dealt with, which is a large scale academic class scheduling. Constraints for these kind of problems are usually expressible in the form of pairs of incompatible objects (e.g., pairs of classes that cannot be assigned to the same room at the same time period). Such incompatibilities are usually embodied through the structure of a graph. Each object (e.g. class) is represented by a node and each incompatibility is represented by an edge joining the two nodes. A coloring of this graph is then simply a partitioning of the objects into blocks (or colors) such that no two incompatible objects end up in the same block. Thus, optimal solutions to such problems may be found by determining minimal coloring for the corresponding graphs. Unfortunately, this may not always be accomplished in a reasonable amount of time.

8.2 Mapping Scheduling onto Graph Coloring

Again, the mapping between a simple case of class scheduling and a graph-based representation goes as follows: The events are represented by the vertices of the graph, and a pair of vertices are joined by an indirected edge if and only if the corresponding events cannot take place at the same time. Scheduling the events subject to the given constraints is therefor equivalent to coloring the corresponding graph such that no two adjacent vertices are of the same color. The determination of the minimum number of intervals of time needed for the schedule is therefore the same as finding the minimum number of colors required for the graph. This is known as the chromatic number of the graph, and its determination for an arbitrary graph is yet unsolved problem.

Let us look at a simple example to illustrate the mapping pictorially. Suppose we have a graph with vertex set V and edge set E, where the ordered pair (R,S) is in E if and only if an edge exists between the vertices R and S. Two vertices in a graph are said to be adjacent if an edge exists between them. Given these sets, the overall GC problem is to partition the vertices into a minimum number of sets in such a way that no two adjacent vertices are placed in the same set. Then, a different color is assigned to each set of vertices, as shown in Figure 8.1. This is known as a proper graph coloring.

In regard to the scheduling problem, our objective is an attempt in finding a schedule satisfying all of the hard constraints and if possible, most of the soft constraints. One essential subset of the hard constraints set is the time-based constraints, and the way to map them onto Figure 8.1 is as follows:

Let each course be represented by a vertex. Two vertices are connected by an edge if and only if there is a reason that the courses they represent may not be offered at

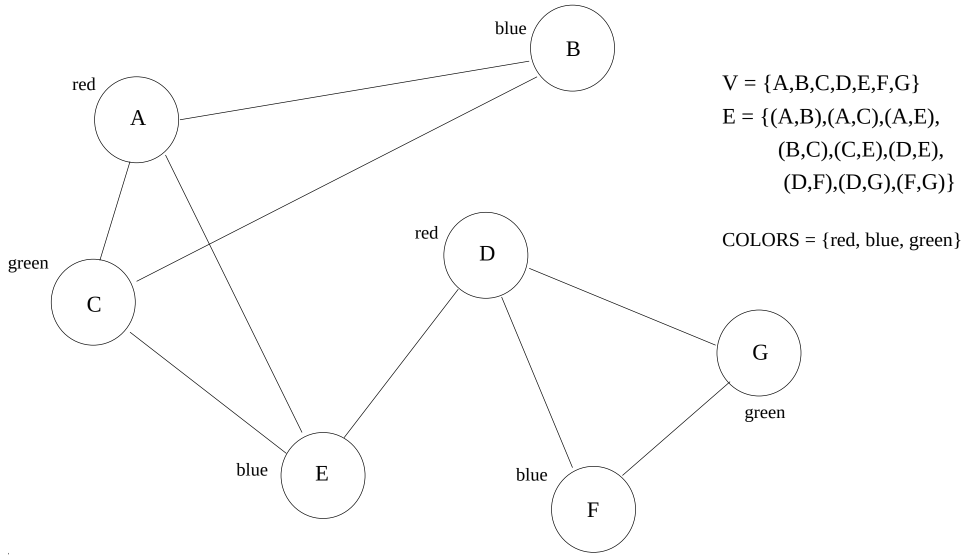

Figure 8.1: A proper graph coloring using three colors.

the same time. Initially, there are two such reasons for vertices to be linked: either the courses they represent are taught by the same instructor, or required by the same set of students. Upon adding in the links between vertices, we assign colors to the graph. Each color represents a different time slot, so every vertex with the same color will be offered at the same time.

Now assume we have, for example, four physics courses: PHY 101, PHY 234, PHY 300, and PHY 300 LAB. The first three which are lectures are taught by the same professor, and the fourth which is a laboratory is taught by another professor. In addition, students enrolled in the lecture section of PHY 300 need also to be enrolled in the lab section of the same course, and vice versa. These time-based hard constraints are represented in the mapping as shown in Figure 8.2. An edge exists between each of the pairs of courses taught by the same professor, since these courses cannot be offered at the same time. Also, an edge exists between the nodes of PHY 300 and PHY 300 LAB, because the same set of students must enroll in both courses. Figure 8.2 also shows a proper coloring with each node of the graph representing a different time slot, so only three different time slots are needed to properly schedule these four courses. PHY 101 and PHY 300 LAB will be offered during the same time slot, PHY 234 will be in a second time slot, and PHY 300 in a third time slot. Note that this system only partitions the courses into groups which will be offered at the same time; it does not assign the actual time of day. This assignment of groups to actual hours of the day can be done in another phase of the scheduling process in which soft constraints are handled.

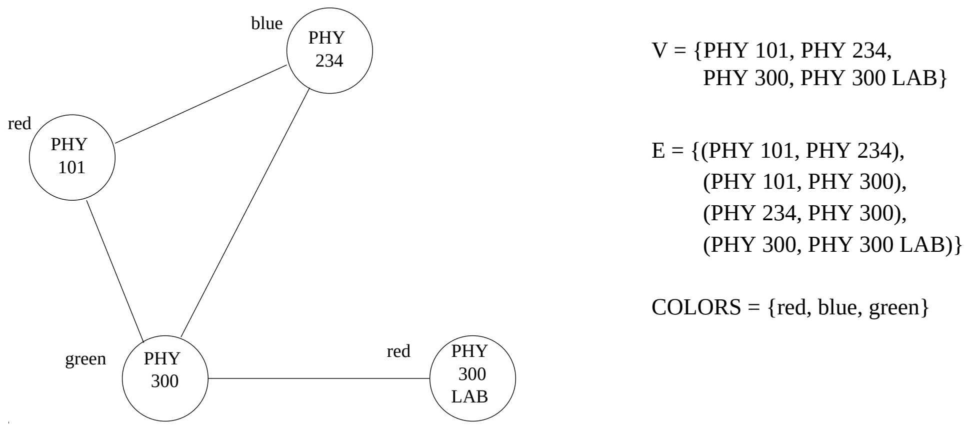

Figure 8.2: A graph representing temporal hard constraints.

8.3 Graph Coloring Algorithm

When dealing with this kind of mapping between course scheduling and graph coloring, the first algorithm that a researcher would investigate would be the simple *saturation algorithm (DSatur, for short)* of Brelaz [Bre79]. As we mentioned in the previous section that the course scheduling problem needs to be simple in order for methods such as *DSatur* to be effective. Before outlining this algorithm, we would like to briefly state one of the characteristics of the problem we have dealt with to show our justification or reasoning for using this algorithm. In addition, we also believe that our observations apply to other similar instance of the problems.

It is often the case that after carrying out the mapping, the representative graphs tend to be loosely connected. One particular reason for this is that any college professor cannot be assigned to teach more than few courses per semester, perhaps maximum of four or five. Therefore, graphs sometimes would show fully-connected clumps of vertices representing those courses. That is partly the reason of having fully (or almost fully) connected components within the overall graph. Also, these components are usually loosely connected among themselves, resulting in a somewhat sparse graphs.

Graph coloring algorithms such as *DSatur* have been shown to work particularly well on sparse graphs. Figure 8.3 and Figure 8.4 show two different versions of the same algorithm.

Let **V** denote the vertex set, and **E** denotes the set of edges of the graph.

- **Repeat** until all vertices are colored
  1. Choose the next vertex **v** from **V** to be colored by selecting the one whose neighbors have already used the most colors. Break ties by choosing the one with the most uncolored neighbors.
  2. Color vertex **v** with the first color in the set of all colors that have not been already used by one of the neighbors, **x** of **V**, such that **v** and **x** are not equal.
- End **Repeat**.

Figure 8.3: DSatur Graph Coloring Algorithm – version (I)

It is interesting to notice that the *DSatur* method is an example of a heuristic procedure for general graphs which is derived from an exact method designed for coloring bipartite graphs (for these we have $|F(x)| \leq 1$ at each step). There are other coloring methods that slightly differ from *DSatur* in their choice of the next node to be colored as well as the color which it will receive.

At the time when we used this algorithm we tweaked with it and made two modifications, both dealing with the available time slots for courses in order to accommodate our problem setup.

Let **V** denote the vertex set, and **E** denotes the set of edges of the graph.

1. Initially take any node with the largest possible number of neighbors and color it with the smallest color.
2. In general, for each **v** of **V**, there is a set **F(v)** of forbidden colors (these may be colors which have already been assigned to neighbors of **v**).
3. At each step, we choose a node **x** of **V** for which the cardinality, —**F(x)**—, of **F(x)** is maximum (highest connectivity with the neighbors), and we color it with the smallest possible color.

Figure 8.4: DSatur Graph Coloring Algorithm – version (II)

1. One of the main constraints of course scheduling is the fixed number of available time slots per week, and the preference to distribute the courses among each of these time slots so that students have many options when selecting courses. Therefore, we modified *DSatur* to select colors for vertices so that, among the previously used legal colors for a vertex in **V**, the one used the fewest number of times previously was selected. This small modification to the algorithm served to distribute the courses more evenly over the time slots. In a typical graph coloring setup, a user usually tries to minimize the number of colors used.

2. The second modification to *DSatur* was necessary due to the fact that time slots for university courses can take different forms. Specifically, the types of time slots dealt with were of two types, although the modification to the graph coloring algorithm can be generalized to more than two types. The first type of time slots, called lecture time slots, were three-credit hour courses that met either three times per week for an hour at a time, two times per week for an hour and a half at a time. The second type of time slots, called laboratory time slots, meet for one three-hour period each week. In order to accurately represent these time slots with colors, two related sets of colors were developed, as illustrated in Figure 8.5.

For example, consider three lecture courses exist which have been colored red, blue, and green. Suppose these colors represent the slots which meet for one hour each on Monday, Wednesday, and Friday at 9:00, 10:00, and 11:00, respectively. Then consider a laboratory course which is taught by the same professor as one of the lecture courses already colored red, blue or green. This lab course may not be assigned the time slot that corresponds to Monday 9-12:00, Wednesday 9-12:00, or Friday 9-12:00, since it would overlap the lecture course. So those three laboratory time slots must be represented by colors which clearly identify the fact that no vertex adjacent to a red, blue, or green lecture course vertex may be given these colors. Therefore, these time slots were labeled with all three conflicting lecture vertex colors, as red-blue-green. However, two different laboratory courses taught by the same instructor could both be scheduled for the red-green-blue time slots, as long as they were on different days. So the red-green-blue laboratory time slots that met on Monday, Wednesday, and Friday were called red-blue-green-Monday, red-blue-green-Wednesday, and red-blue-green-Friday, respectively, to distinguish them from one another.

Consequently, due to our use of two sets of timeslots (lecture and laboratory timeslots) and also due to our criterion of scheduling one set prior to the other, we had to modify the method to color the lecture-of-course set of vertices first, followed by the lab-of-course set of vertices. Without this modification there is no distinction between various sets of vertices. Also, during the scheduling process we needed to do "re-coloring" of the graph that was constructed to get a legal schedule.

8.3.1 Spatial Hard Constraints

Once the modified graph coloring algorithm has partitioned the vertices into time slots, a room must be assigned for each course in each time slot. One constraint that must be fulfilled is that no course may be assigned to a room which has a smaller capacity than the maximum enrollment of the course. Additionally, each

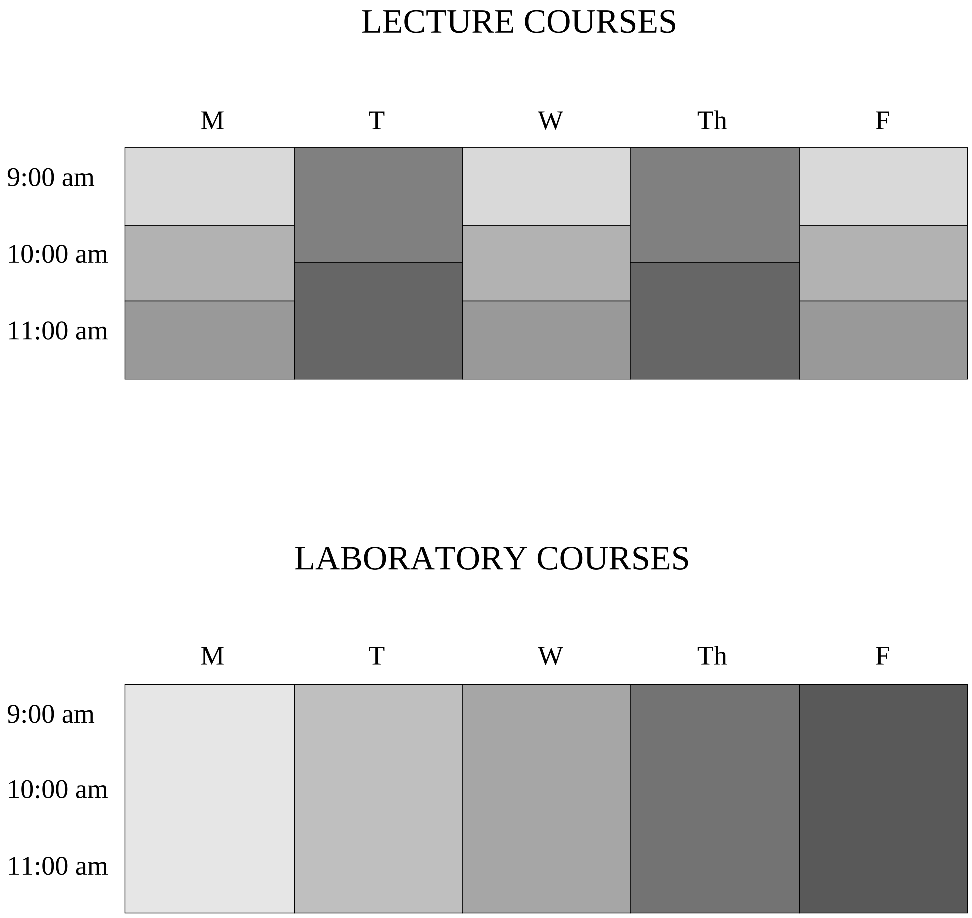

Figure 8.5: Time slots belonging to two overlapping sets.

room available on campus is of a particular type, such as lecture, seminar, laboratory, etc. Each course requires one of these types of rooms. Furthermore, some rooms are reserved by particular departments for their courses only, such as a chemistry laboratory room, while other rooms are general usage rooms, for any department to use. An algorithm was developed to assign all of the courses which have been assigned to the same time slot to appropriate rooms, assigning courses to the rooms reserved by department first, then the general purpose rooms, always considering the largest capacity rooms and largest courses first.

However, this simple algorithm alone is not sufficient. Consider the case where 25 courses of maximum enrollment 100 are assigned to the same time slot. Suppose that only 20 rooms of capacity 100 or greater exist on campus. There is no way to create a legal schedule of courses with all 25 courses in the same time slot. The solution to this problem was to re-color the graph, having added edges between each of the vertices representing "leftover" courses, that is, those who did not get assigned a room, and then re-assign rooms. Adding the edges between these vertices ensures that they will not end up assigned to the same time slot, and therefore will not vie for the same rooms at the same time.

8.3.2 Structure of Code

The structure of the code, when the algorithms for both temporal and spatial constraints are included, is shown in Figure 8.6. This code produces a full legal solution to the course scheduling problem, that is, one that satisfies all of the hard constraints. Ideally, this solution would then be slightly altered or fed into the next phase of simulation to satisfy the maximum number of the soft constraints.

```
CODE STRUCTURE

  Input course data, inserting links between
     co-requisite courses
  Input instructor data, inserting links between
     courses taught by the same professor
  Input room data, separating reserved usage
     rooms from general usage rooms

  REPEAT UNTIL number of conflicts = 0
      Color lecture-course vertices
      Color laboratory-course vertices
      Assign rooms, adding links to the
         graph where conflicts exist
  END REPEAT
```

Figure 8.6: The overall structure of the course scheduling algorithm.

Chapter 9

Airline Crew Scheduling versus Course Scheduling

9.1 Complexity of Course Scheduling

...

...

9.2 Airline Crew Scheduling

Airline crew scheduling is classified as a resource allocation problem, where a given flight schedule is to be covered by a set of crew *rotations*. Each rotation consists of a a connected sequence of flights or *legs*, each starting and ending at a given home base or *hub*. Then the objective is to minimize the total waiting time of the crew subject to a number of constraints on the rotations. Its topological structure and the restrictions imposed on it is quite similar to a multi-task phone routing structure.

Perhaps, one particular approach that has been used in tackling this problem is first to convert it into a set covering problem, by

1. generating a large number of legal rotation templates, and

2. seeking a subset of these templates that precisely covers the entire flight schedule.

Then, solutions to the set covering problem are found with conventional methods such as linear programming or other methods such as feedback neural networks.

This will have a big disadvantage and that is, on one hand, from a computational point of view, an exhaustive generation of rotations is not feasible for large real world problems; on the other hand, a non-exhaustive generation will only cover a fraction of the solution space.

One other approach is to proceed in two phases (see Lagerholm at. el EJOR 1998):

1. the full solution space is narrowed down by using a reduction method that removes a large part of the sub-optimal solutions; then

2. a mean field annealing approach is applied to handle the topology, leg-counting, etc.

It has been determined that the computational requirement for random artificial problems with resemblance to real-world situations grows in the order of N_f^3 where N_f is the number of flights. So, given a schedule in terms of a set of N_f flights (let say per week), with specified times and airport departure and arrival, a crew is to be assigned to each flight such that the total crew waiting time is minimized subject to the following restrictions:

- Each flight crew must follow a connected path, or *rotation*, starting and ending at the *hub*.

- The number of flight *legs* in a rotation must not exceed a given upper bound, i.e. L_{max}.

- The total duration (flight + waiting time) of a rotation is similarly bounded by a maximum time, i.e. T_{max}.

In a real-world case there are many constraints to be satisfied, but in general the above three are the most crucial and difficult ones. Furthermore, without them the problem would reduce to that of minimizing waiting times independently at each airport – the *local problems*; which can be solved exactly in polynomial time; e.g. by pairwise connecting arrival flights with subsequent departure flights. Therefore, without the global structural requirements, the crew scheduling problem is not much of a challenge.

9.2.1 Crew Scheduling vs. Course Scheduling

The most noticeable difference between academic course scheduling and crew scheduling is in the topological structures. Course scheduling problem has a non-Euclidean global structure while the structure (or the graphical representation) of crew scheduling is quite similar to a multi-task phone routing structure. Neural networks based methods such as the abovementioned mean field annealing approach has been shown by several authors to be unsuited for tackling non-Euclidean problems [Sim92].

The other distinction is in the duration. Typically, a real-world flight schedule has a basic period of about one week; while the duration of a university course schedule is an academic semester or a quarter. Also, in crew scheduling, airlines restrict crew members in spending double overnights in any of the non-*hub* points of the flight; and to take a rest for a certain number of hours after each duty period. These restrictions are considered from a constraint point of view to be hard so it would need to be satisfied for the final schedule. On the other hand, the analogous set of constraints dealing with students and professors in course scheduling are generally treated as soft or medium constraints. In general, for course scheduling we do not have a well defined objective function to optimize; there are many requirements which occur as constraints in the problem and a collection of preferences for students and professors, that are partially formulated as low priority constraints.

9.3 To Be Done...

Some of Geoffrey's suggestions to be taken into account in the thesis.

- A section giving the complexity of the problem. This would include runtime of various algorithms used, number of variables, size of the overall problem and data, etc.

- A chapter explaining why some optimization problems can also be tackled by methods such as used in this thesis, particularly the annealing approaches... While others are not suitable. For example, Genetic Algorithms are known to be quite good in tackling certain problems but not so good in tackling others. It would be quite useful to shed some light on the reasons behind this. Also, it would be interesting to have a section on the reasons why the neural net approach does not do well on scheduling problems.

Chapter 10

On Performance of Local Search Algorithms

Chapter 11

Glossary and Definitions

Definitions of some of the key terms appearing in the thesis.

- **Fitness Function**: Fitness function is a mathematical equation that describes the relative value of each member of a family of objects with respect to some criterion. The fitness function is maximized or minimized (depending upon the criterion) to find the best object in the family.

- **Landscape Analysis**: Landscape analysis is a mathematical process used to search for the best solution to a multi-variable optimization problem. It utilizes a fitness function. Within the search space (i.e., a plot of all the possible solutions), the fitness of a particular solution can be shown graphically as its height. The resulting "landscape" will have peaks in the regions of the search space that contain better solutions; further analysis of the peak regions can be used to refine the results and achieve even more precise solutions.

- **Fitness landscape**: A fitness landscape is a particular kind of representation of a large space of "configurations". For example, this space could be "The space of all possible routes through 100 given cities" or "The space of all possible genotypes of a given length". A fitness landscape requires both a *fitness function* assigning a value (usually in real numbers) to each *configuration*, and a *neighborhood relation* amongst the configurations.

- **Multi-Objective Optimization**: Is the task of maximizing or minimizing a number of criteria simultaneously. Multi-objective optimization is more difficult than most kinds of optimization problems because the trade-offs between the multiple objectives may be unknown.

- **Simulated Annealing**: A mathematical technique for general optimization problems. The same comes from the physical process of annealing, during which a material is first heated and then slowly cooled. During annealing, the component atoms of a material are allowed to settle into a lower energy state so that a more stable arrangement of atoms is maintained throughout the cooling process. SA has been applied with success to a wide variety of optimization tasks.

- **Optimization problem**: A search problem in which we wish to know, in some given domain, the minimal (or maximal) value of a given function. E.g. find $min_{\sigma \in \Sigma} f(\sigma)$, where $f : \Sigma \rightarrow \mathcal{R}$.

- **Spin Glass**: Spin glasses are simplified models from physics describing dilute magnetic alloys. They define an energy function (or equivalently a fitness function) over tiny magnetic dipoles called spins. Spins may point either up (+1) or down (-1). For every configuration of spins (e.g., +1-1+1+1-1 for a configuration of 5 spins), the energy function assigns that configuration's energy by summing up the interactions between spins. many aspects of spin glasses are utilized by physicists and they comprise a well-understood family of fitness landscapes.

- **Entropy**: A mathematical measure of the amount of disorder in a system.

- **Phase transition**: Loosely a qualitative change in behavior resulting from the smooth variation of a parameter. For example as we reduce the temperature, water undergoes a phase transition when it freezes: change from water into ice.

Bibliography

[Abr91] D. Abramson. Constructing school timetables using simulated annealing: sequential and parallel algorithms. *Managment Science*, 37:98–113, 1991.

[ADK00] D. Abramson, H. Dang, and M. Krishnamoorthy. Simulated annealing cooling schedules for the school timetabling problem. *Asia-Pacific Journal of Operation Research*, 2000.

[AF89] J. Aubin and J. A. Ferland. A large scale timetabling problem. *Computers and Operations Research*, 16:67–77, 1989.

[AF91] A. V. B. Aiyer and F. Fallside. A hopfield network implementation of the viterbi algorithm for hidden markov models. In *Proceedings of the International Joint Conference on Neural Networks*, pages 827–832, 1991.

[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.

[AKvL97] E. H. Aarts, J. Korst, and P. J. van Laarhoven. Simulated annealing. In Emile Aarts and Jan Karel Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley-Interscience Series in Discrete Mathematics. John Wiley and Sons, Chichester, UK, 1997.

[AL97] Emile Aarts and Jan Karel Lenstra, editors. *Local Search in Combinatorial Optimization.* Wiley, Chichester, England, 1997.

[ANF90] S. V. B. Aiyer, M. Niranjan, and F. Fallside. A theoretical investigation into the performance of the hopfield model. *IEEE Transcations on Neural Networks*, 1(2):204–215, June 1990.

[Bre79] D. Brelaz. New methods to color vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[Bru90] J. Bruck. On the convergence properties of the hopfield model. In *Proceedings of the IEEE*, volume 78, October 1990.

[BV55] G. E. P. Box and P. V. Voule. The exploration and explotation of response surfaces: An example of the link between the fitted surface and the basic mechanism of the system. *Biometric*, 11:287–323, 1955.

[CG83] M. A. Cohen and S. Grossberg. Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Transcations on Systems, Man and Cybernetics*, 13(5):813–825, September/October 1983.

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

[Coo71] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[DLS93] R. Diekmann, R. Luling, and J. Simon. Problem independent distributed simulated annealing and its applications. In R. V. V. Vidal, editor, *Lecture Notes in Economics and Mathematical Systems – Applied Simulated Annealing*, volume 396. Springer, Berlin, Germany, 1993.

[EDK+91] S. P. Eberhart, D. Daud, D. A. Kerns, T. X. Brown, and A. P. Thakoor. Competitive neural architecture for hardware solution to the assignment problem. *Neural Networks*, 91:431–442, 1991.

[EFC96] M. A. S. Elmohamed, G. Fox, and P. Coddington. Course scheduling using mean-field annealing, part i: algorithm and part ii: implementation. *NPAC Technical Report*, SCCS(782), 1996.

[EFC97] M. A. S. Elmohamed, G. Fox, and P. Coddington. Academic scheduling using simulated annealing with a rule-based preprocessor. *NPAC Technical Report*, SCCS(781), 1997.

[EL87] H. A. Eiselt and G. Laporte. Combinatorial optimization problems with soft and hard requirements. *Journal of Operational Research Society*, 38(9):785–795, 1987.

[FHL96] J. A. Ferland, A. Hertz, and A. Lavoie. An object-oriented methodology for solving assignment-type problems with neighborhood search techniques. *Operations Research*, 44:347–359, 1996.

[FL92] J. A. Ferland and A. Lavoie. Exchange procedures for timetabling problems. *Discrete Applied Mathematics*, 35:237–253, 1992.

[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.

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

[GP92] A. H. Gee and R. W. Prager. Polyhedral combinatorics and neural networks. *Neural Computation*, May 1992.

[Gro88] S. Grossberg. Nonlinear neural networks: Principles, mechanisms and architecture. *Neural Networks*, 1(1):17–61, 1988.

[GSP89] L. Gislen, B. Soderberg, and C. Peterson. Teachers and classes with neural nets. *International Journal of Neural Systems*, 1:167–176, 1989.

[GSP92] L. Gislen, B. Soderberg, and C. Peterson. Complex scheduling with potts neural networks. *Neural Computation*, 4:805–831, 1992.

[GW93] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for sat. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages xxii+869, 28–33, 1993.

[GW94] I. P. Gent and T. Walsh. The sat phase transition. In *ECAI94, Proceedings of the 11th European Conference on Artificial Intellidence*, pages xvi+832, 105–109, 1994.

[HHW96] T. Hogg, B. Huberman, and C. Williams. Phase transitions and the search space. *Artificial Intelligence*, page 81, 1996.

[HKP91] J. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwood City, CA, USA, 1991.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[Hop82] J. J. Hopfield. Neural networks and physical syetms with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences USA*, volume 79, pages 2554–2558, April 1982.

[Hop84] J. J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. In *Proceedings of the National Academy of Sciences USA*, volume 81, pages 3088–3092, May 1984.

[HRSV86] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. In *Proc. of the IEEE International Conference on Computer Aided Design (ICCAD)*, pages 381–384, 1986.

[HT85] J. J. Hopfield and D. W. Tank. Neural computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[HT86] J. J. Hopfield and D. W. Tank. Computing with neural circuits: a model. *Science*, 233:625–633, 1986.

[JAMS89] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation, part i (graph partitioning). *Operations Research*, 37:865–892, 1989.

[JAMS91] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation, part ii (graph coloring and number partitioning). *Operations Research*, 39:378–406, 1991.

[JM97] D. Johnson and L. McGeoch. The traveling salesman problem: A case study in local optimization. In *Local Search in Combinatorial Optimization*, Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley and Sons, Chichester, UK, 1997.

[Kar72] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[KGV83] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[Kir84] S. Kirkpatrick. Optimization by simulated annealing: Quantitive studies. *Journal of Statistical Physics*, 34:976–986, 1984.

[KS87] I. Kanter and H. Somplinsky. Graph optimization problems and the potts glass. *Journal of Physics A*, 20:L673–L679, 1987.

[LG92] J. E. Lewis and L. Glass. Nonlinear dynamics and symbolic dynamics of neural networks. *Neural Computation*, 4(5):621–642, 1992.

[Lis93] R. Lister. Annealing networks and fractal landscapes. In *Proceedings of IEEE International Conference on Neural Networks*, volume I, pages 257–262, March 1993.

[MEY95] S. Miner, M. A. S. Elmohamed, and H. W. Yau. Optimizing timetabling solutions using graph coloring. In *NSF REU Program – 1995 Reports*. NPAC, Syracuse University, Syracuse, NY, 1995.

[Mou84] O. G. Mouritsen. *Computer Studies of Phase Transitions and Critical Phenomena*. Springer-Verlag, Berlin, Germany, 1984.

[OB90] R. G. Ogier and D. A. Beyer. Neural network solution to the link scheduling problem using convex relaxation. In *Proceedings of the IEEE Global Telecommunications Conference*, pages 1371–1376, 1990.

[OvG89] R.H. J. M. Otten and L. P. P. P. van Ginneken. *The Annealing Algorithm*. Kluwer, Norwell, MA, 1989.

[PCSD89] C. Petrie, R. Causey, D. Steiner, and V. Dhar. A planning problem: Revisable academic course scheduling. *MCC Technical Report*, ACT-AI(020), June 1989.

[PR88] R. G. Parker and R. L. Rardin. *Discrete Optimization*. Academic Press, Inc., 1988.

[PS89] C. Peterson and Bo Soderberg. A new method for mapping optimization problems onto neural nets. *International Journal of Neural Systems*, 1:3–22, 1989.

[Sim92] M. W. Simmen. Neural network optimization. *PhD Thesis*, 1992.

[SK91] P. N. Strenski and S. Kirkpatrick. Analysis of finite length annealing schedules. *Algorithmica*, 6:346–366, 1991.

[SK93] B. Selman and H. A. Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Chambery, France, 1993. IJCAI.

[SKC96] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Proceedings of the 2nd DIMACS Implementation Challenge: Cliques and Coloring and Satisfiability*. American Mathematical Society, 1996.

[SLM92] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the American Association of Artificial Intelligence (AAAI)*, pages 440–446, San Jose, CA, 1992. AAAI.

[Sor91] Gregory B. Sorkin. *Theory and Practice of Simulated Annealing on Special Energy Landscapes*. Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1991.

[SY91] A. A. Schaffer and M. Yannakakis. Simple local search problems that are hard to solve. *SIAM Journal on Computing*, 20:56–87, 1991.

[TCP88] G. A. Tagliarini, J. F. Christ, and E. W. Page. A neural-network solution to the concentrator assignment problem. In D. Z. Anderson, editor, *Neural Information Processing Systems*, pages 775–782. American Institute of Physics, New York, 1988.

[TD95] J. Thompson and K. Dowsland. General cooling schedules for simulated annealing based timetabling systems. In *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, Edinburgh, Scotland, 1995.

[TTR91] L. Tarassenko, J. N. Tombs, and J. H. Reynolds. Neural network architectures for content-addressable memory. In *IEE Proceedings, Series F*, volume 138, pages 33–39, February 1991.

[Vid93] R. V. V. Vidal, editor. *Applied Simulated Annealing*. Lecture Notes in Economics and Mathematical Systems. Springer, Berlin, Germany, 1993.

[vLA87] Peter J. M. van Laarhoven and Emile H. L. Aarts. *Simulated Annealing: Theory and Applications.* Reidel, Dordrecht, NL, 1987.

[Whi84] S. R. White. Concepts of scale in simulated annealing. In *Proceedings of the IEEE International Conference on Circuit Design*, pages 646–651, 1984.

[WP88] V. Wilson and G. S. Pawley. On the stability of the tsp problem algorithm of hopfield and tank. *Biological Cybernetics*, 58:63–70, 1988.

[Wu82] F. Y. Wu. The potts model. *Reviews of Modern Physics*, 54:235–268, 1982.