THE UNIVERSITY OF CALGARY

INVESTIGATIONS INTO THE MASTER TIMETABLING PROBLEM

by

COLIN J. LAYFIELD

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

SEPTEMBER, 1998

Abstract

Timetabling is a varied and interesting topic in the field of operational research. The problem of scheduling courses for departments in a university environment, like the University of Calgary, is no exception. Two methods are examined in order to try to solve this problem. First, a tailor made *heuristic* is developed. Next, a *genetic algorithm*, a more generic problem solving metaheuristic, is also developed and applied towards this problem. The Biology/Chemistry department's courses are used as a challenging test bed for these two approaches and the results are compared with the actual schedule used in the 1994-1995 academic year. Both methods could create schedules better than the actual schedule with the heuristic creating the better solutions between the two methods.

Acknowledgements

I would like to extend my gratitude to my supervisor Dr. Colijn. I have worked with him for many years and I am grateful for the insight he has given me into scheduling as well as the time he has spent in helping me turn this project into a successful thesis.

I would also like to thank Dr. Gary Krivy for employing me several years ago to work on the final examination timetabling system. Dr. Krivy also funded the initial work into the master timetabling problem. Without his initial funding and support this project would have never happened.

To my parents, Doreen and Jim Layfield, I would also like to state my appreciation for their never ending support towards my goal of finishing this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Scheduling Problems

Scheduling and timetabling problems surround us in everyday life. From bus and train timetables for city commuters to final examination timetables for university or college students it is a common problem that can prove to be difficult to solve. It can be the case that even finding a moderately good solution is very difficult. According to [Dav91] scheduling is difficult for two reasons:

1. It is a computationally complex problem that is intractable. This means that an optimum solution cannot usually be found in polynomial time. This is because examination of the complete problem solution set is not practical and it is difficult to determine whether a solution is an optimum without examining a large part of the solution set.

2. Scheduling problems are often complicated by the details of the particular scheduling task; for example the real world constraints that must be adhered to in order to make the schedule or problem solution practical.

Since these are problems for which it is difficult to find good solutions, it is worthwhile to see if any improvements can be made over manually produced solutions. An example of why this may be the case can be found in the *exam scheduling* problem. When a final examination schedule is generated at an educational institution, such as a university, it is desirable to have an exam schedule that is both as short as possible

as well as as fair as possible to the students. The reason for a short schedule is often dictated by institutional requirements[1]. The issue of creating a schedule that is *fair* to the student should be a straightforward one. A examination scheduling system should, for example, try to reduce the number of exams that a student has to write back-to-back as well as minimizing the number of exams the student will have to write in a day. Exam schedule generation is a well known NP-Hard problem but work has been done to try to optimize these schedules to make them better for students and faculty. The examination system currently used at the University of Calgary is an example of one such system that has proven to be successful in practice from both the point of view of the student and the Registrar's office with the criteria outlined above as well as other more in depth topics [Lay93, CL95b, CL95a].

1.2 The Master Timetable Problem

The main topic of the work done in this thesis is solving the *Master Timetabling Problem* (MTTP). The first formulation of this type of problem is credited to [Got63]. Gotlieb's formulation was for Toronto high-schools but nevertheless shows that this problem has been around for some time. The schedules that will be dealt with here are considerably larger and more complex. In the MTTP, a set of courses, C, a set of timeslots, T, whose members can be assigned to courses, and a set of rooms, R, into which courses can be placed, are given. From these sets of data the problem is to produce a timetable such that every course is assigned both a timeslot and room. These assignments must satisfy some criteria that make the schedule *feasible*[2]. Specifically, the problem being looked at is the generation of a master timetable at the University of Calgary for various departments. The MTTP at the University of Calgary is different from similar problems at certain other institutions in that the master timetable schedule is *student* based as opposed to *course* or *class* based. The

[1] It is almost always the case that the institution has a fixed number of days (exam periods) that can be used so the schedule *must* fit into these periods and, therefore, it is beneficial to make as short a schedule as possible so as to leave flexibility for the examination officer to manipulate the schedule if necessary.

[2] These criteria are designed to capture a number of requirements, including, for example, that required courses in a given year of a program are to be scheduled in such a way that students can enroll in them without time conflicts.

distinction here is that with student based schedules the student has the option of choosing which courses he/she wants to take so the combinations of courses available to students are much larger. In other words the instituation tries to meet the needs of the students. With course or class based schedules the student moves as part of a class from course to course. There may be some variation in the courses available but the selection is far more limited from the point of view of the student than what is available in the student based schedule. An example of a class based schedule can be found at the Southern Alberta Institute of Technology's timetable.

It is interesting to observe that if restrictions are taken off the class based timetable generation process then the timetable problem turns into a simple assignment problem [DeW71b]. An example of this would be if the only restriction during timetable construction considered is that no teacher or course is to be in two places at the same time.

1.3 Algorithmic Approaches

1.3.1 Background

The timetabling problem has been investigated for a long time and many approaches exist to try to deal with this problem. Since NP-Hard problems are difficult to solve and since writing algorithms to find an exact solution is not practical, other methods are often used to deal with these problems. Examples of these are *approximation algorithms* or *heuristics* which have been applied to such difficult problems. Examples of other approaches to timetabling include using techniques such as *network flow methods* (which de Werra successfully applied to school timetables [DeW71b]), *integer-programming* and *constraint based programming* [CKLW95, CLR90].

This research concerns two different approaches to the MTTP problem. The first approach is a *heuristic approach* involving a limited amount of *backtracking*. The second approach is known as a *genetic algorithm* or an *evolutionary programming* approach. The heuristic backtracking approach was developed from a base of summer work done in conjunction with the Registrar's office at the University of Calgary with regard to the computer generation of the master timetable for various departments. The genetic algorithm approach is also examined due to a personal interest in the

topic of GA's by the author.

1.3.2 Heuristics

The Heuristic Approach (HA) often works trying to solve the problem incrementally by making, it is hoped, intelligent choices along the way to improve the quality of the solution found. When a technique such as this runs into trouble (for example, no feasible timeslots are available for a certain course due to poor choices made earlier in the process) a technique known as backtracking is invoked where the algorithm goes back to some previous state and tries a different choice to the one that may have caused the initial problem. The quality of a heuristic backtracking approach is generally quite good since the approach is *custom made* for the problem. The downside to this method is that it is not very robust and unless good programming practices and foresight are applied it may be difficult to modify the algorithm to take into account other constraints that may be added to the problem at a later date.

1.3.3 Genetic Algorithms

Genetic Algorithms (GA's) use a much different approach than that of HA's. The GA approach works based on an analogy to natural processes. The idea is based on the concept of *survival of the fittest* and of *evolution*. The algorithm starts with a population of potential *solutions* to the problem, in this case the MTTP. This population undergoes a reproduction stage where characteristics of many solutions in the population are combined through a mating process. The probability of various solutions mating to produce offspring for the next generation is biased towards individuals that are *more fit*, meaning that better solutions are more likely to participate in reproduction. The probabilistic choice made helps to ensure that more of the solution space is examined and helps prevent the algorithm from getting stuck in a local optimum. The population of solutions should start to become more fit generation after generation as the good solutions are more likely to survive and contribute to future generations. Good characteristics from solutions are combined with binary operators such as *crossover* whilst diversity in the population is maintained with unary operators such as *mutation* which are taken directly from their biological context [Cam93].

Good textbooks and articles to explore the fundamentals of GA's can be found in [Dav91, Mic92, SP94, Ree93].

More exploration of the background behind these methods will be dealt with in Chapter 3.

1.4 Goals

The goal of this work is to find a good method for solving the MTTP. Both GA's and the HA are implemented and considered and both, as one would expect, have their advantages and disadvantages. The work and results found here will also be used as a continuation point for the development of a timetabling system for use at the University of Calgary as with larger classes and dwindling resources[3] efficient use of them will become even more important in the future.

1.5 Overview

The next chapter of this thesis will deal with a further exploration of the problem itself and the data supplied. Chapter 3 will deal with a much more in depth background discussion of Heuristics as well as Genetic Algorithms. Chapter 4 covers the heuristic implementation of an algorithm to generate MTTP solutions. Chapter 5 will delve into the application of Genetic Algorithms to the MTTP. Chapter 6 will discuss the results that were produced with both methods as well as a discussion of the implications of the differences of the two approaches and results. The final chapter will reiterate the findings and methods used and contain a discussion of further research that may be beneficial.

[3] Both financial and human resources!

Chapter 2

The Master Timetable Problem

Every year at most universities the *Master Timetable* or *Lecture Timetable* is published which outlines all of the courses available to students in the upcoming term. It lists the courses (lectures, laboratories, tutorials and/or other instruction types), the times they are available (often courses have multiple lectures, laboratories or tutorials), and the rooms in which they are held. Each institution, it would seem, has its own method in which the timetable is defined as well as possessing different course structures and flexibility (for example the aforementioned student and course based approaches).

The basic element of the Master Timetable problem is the set of *courses* or *course sections* to be scheduled, $C = \{c_1, c_2, \ldots, c_n\}$. Each course section (or, sometimes, just *section*) has various properties associated with it such as its duration or special rooms it needs to use. Each of these sections must have both a time and room assigned to it from the set of all rooms, $R = \{r_1, r_2, \ldots, r_p\}$, and the set of all timeslots, $T = \{t_1, t_2, \ldots, t_q\}$. The set of rooms is, obviously, the rooms available to the scheduler to place course sections in. The set of timeslots is the collection of days and times when the sections can be held. At many institutions a standard lecture must have 150 minutes a week of lecture time. For example, a timeslot might take this into account by having a 50 minute lecture at 11:00AM on Monday, Wednesday and Friday or it could alternatively be put into two 75 minute length lecture periods held on Tuesday and Thursday. The two examples shown above are common timeslots that are available in the set of timeslots, T. A Master Timetable is created when, for each

section, an *assignment* is made. An assignment is a triple (c, t, r) where $c \in C$, $t \in T$ and $r \in R$. This definition of the MTT problem was taken from [RCF94b, CRF94b] and is used in both of the approaches examined.

The University of Calgary uses a student-based approach to the programmes that are offered. This, in some sense, makes the problem more difficult as it is desirable to have as many courses non-conflicting as possible since it may be the case that a student wants to take various combinations of courses as options[1]. The way the course sections are set up and how they function also deserves further detailed explanation as it probably differs to some extent from most other Universities.

2.1 Course Sections

The MTT is made up entirely of *course sections* or just *sections*. Each course has at least one lecture section[2], possibly more. Each course may also, but not always, have laboratory sections as well as tutorial sections. These different types of sections will be referred to as *instruction types* or *section types*. A good example would be the first year Computer Science course CPSC 203. The course usually has two or three lecture sections, say three, (since it is popular as a science option with non Computer Science majors) and multiple lab sections, say 10, (many of their assignments are to be done in a PC lab and a lab cannot usually hold as many students as a lecture hall). This course has no tutorials at all. The set of sections to be scheduled would look as follows[3]:

$$C = \{\ \text{CPSC203L01, CPSC203L02, CPSC203L03, CPSC203B01,}$$
$$\text{CPSC203B02, ..., CPSC203B10}\ \}$$

[1] This is usually the case for junior courses as, obviously, courses like advanced nuclear physics are unlikely to be taken as an option by a Kinesiology major.

[2] There are a few exceptions to this rule as some courses have no lectures. The number of these is sufficiently small such that they can be ignored however.

[3] At the University of Calgary the courses are represented in the calendar by their abbreviation followed by a 'L', 'B', or a 'T' which represent a Lecture, laB or Tutorial followed by the course section number.

Notice that each lecture and lab section is a distinct element in the set C. The reasoning for this should be clear as each section must be scheduled individually. The sets of timeslots and rooms available to these sections should not be the entire sets T and R. The lab sections, for example, would have to be held in special rooms that are equipped with PC's that are running the software needed to complete the assignments required so there would be no need to make a lecture hall available for a CPSC203 lab section. The lecture halls needed to place the lectures for this course should be fairly large because the enrollment for each lecture, in this case, can exceed 100 students with ease so having a small tutorial room flagged as available to this section would also be a pointless operation.

2.1.1 Room and Timeslot Categories

The purpose of *room categories* and *timeslot categories*, as will be seen, is to enable the specification of what rooms and timeslots *should* be made available to each course section. Each section, C_i, will have a *timeslot category*, T_i, as well as a *room category*, R_i, such that $T_i \subseteq T$ and $R_i \subseteq R$. This provides for an easy method to specify the rooms and timeslots that should be available to the sections individually while leaving out any timeslots or rooms that are not applicable for the section.

With these facts in mind the conceptual data that is to be made available to a scheduling system will consist of the following:

C = Set of *Course Sections* or *Sections* that are to be scheduled

T = Set of *TimeSlots* in which courses can be held.

R = Set of *Rooms* in which courses can be held.

The additional families of sets $TC = \{T_1, T_2, \ldots, T_n\}$ and $RC = \{R_1, R_2, \ldots, R_m\}$ are also added where RC is a set of *Room Categories* and TC is a set of *Timeslot Categories* as explained above.

2.2 Constraints

As mentioned earlier, in practice, course sections cannot use just any timeslot. It may also make little sense to have course sections recorded as having access to just any room. For example Biology labs can sometimes take up to 4 hours at a time while an English lecture might be scheduled to take only 150 minutes of lecture time per week. In fact, the University of Calgary standard policy for lecture time is 150 minutes a week so clearly a 4 hour timeslot would not be useful for lecture time. Also rooms cannot normally be booked simultaneously (two classes using the same room at the same time). Some rooms also have specialized purposes so that only certain courses can make use of them. An example of this would be a physics lab classroom which would only be useful for a physics lab class because of the special equipment it may have. These constraints are fairly obvious and most institutions would have similar constraints that must be honoured. The concepts of Timeslot Categories and Room Categories effectively address this problem.

Other constraints that are perhaps more dependent on the institution in question would be combinations of courses that have to be available for students to take. For example, in the University of Calgary Calendar [Cal93] on page 317 a table is given that shows the recommended sequence in the first two years for both the honours and major programmes in Computer Science[4]. The table for second year Computer Science (CPSC) can be seen in Table 2.1.

Fall Term	Winter Term
Computer Science 331	Computer Science 333
Computer Science 321	Computer Science 313
Computer Science 355	Computer Science 357
Option	Philosophy 379
Option	Option

Table 2.1: Recommended Course Sequence For 2nd Year CPSC.

From Table 2.1 it can be seen that, in the fall term for second year Computer Science, the students usually take computer science 331, 321, and 355 along with two

[4]Most of the test data was obtained in 1993 so it is convenient to keep using it without updating it for more recent data. Not much of essence has changed in the calendar since then.

options. In the winter term the students usually take Computer Science 333, 313, 357, Philosophy 379 and one option. When a master timetable is generated students must be able to register in the lectures, labs and tutorials of all courses that should be taken during the same term So in the fall term, Computer Science 331, 321 and 355 cannot have conflicting lecture times and must have laboratory time available such that students registered in all three courses can also enroll in applicable lab times as well as any tutorials the courses may have. Clearly any method used to generate the master time table will have to take into account constraints such as these as this is the essence of a good timetable. Students in various disciplines must be able to take the courses they need to have each term. These constraints are often called *edge constraints* due to their similarity to simpler timetabling problems, like examination scheduling, that employ graph colouring methods [CRF94b]. These are also, usually, the most common constraints to be placed on scheduling problems such as this.

Another similar issue that may arise comes from programmes of study that may not have recommended courses to take. An example would be the English degree. There is great flexibility in the courses that can be taken and the order in which they are taken, so generating a list of potential constraints can be quite lengthy and inefficient. A possible option in this case would be to look at *historical enrollments* and take that into consideration when making the MTT. This way, courses that happen to be taken together by a substantial number of students can have a constraint between them added such that they are not scheduled at the same time in order to reflect this historical enrollment information. This, unfortunately, makes the assumption that the historical data represents a good schedule for departments such as these. Another option could be, if resources are available, to augment the historical data with a student survey to determine what courses the students themselves would like to be able to enroll in during the same term. Consultation of the various department heads on what courses would be 'natural' to take together is yet another option.

Conflicts between courses due to information found in the calendar (like Table 2.1), or historical data if no such table is supplied, will be referred to as *Calendar Conflicts* or *Calendar Constraints*.

2.2.1 Other Factors

Other factors can be taken into account: ensuring the instructors teaching the sections are not assigned overlapping timeslots; trying to minimize the distance between sections for lecturers; trying to group sections in the same building, if possible; and so on. In this research some were taken into account, others not. There's a variety of reasons for this. In the case of instructor conflicts, the information often isn't available when the timetable is constructed, thus making it impossible.

To sum up this section, the goal of the research is to be able to generate a master timetable that is better than the ones used in the past (which, it is hoped, implies the techniques used can be used for future master timetable generation). This involves taking a set of course sections and assigning each section (which may be a lecture, lab or a tutorial instruction type) a valid room and timeslot. While this is being carried out, care must be taken that certain obvious constraints are taken into account, for example, not scheduling two courses in the same room at the same time. Other constraints such as the calendar conflicts must also be taken into account by trying to ensure all sets of courses that should be available to a student are not scheduled in conflicting timeslots, if possible.

The next chapter will discuss the two methods that are to be applied to this problem. They are a backtracking heuristic method and genetic algorithms.

Chapter 3

Background

3.1 Heuristics

Since scheduling problems like the Master Timetable problem are widely known to be NP-Hard, the only practical approach to them is to find a good selection of heuristics to approximate good solutions. There are two types of heuristics that are applied in this thesis to the MTT problem and background information on both of them will be discussed. The first technique is the more traditional backtracking heuristic approach. The second technique that will be applied is a metaheuristic referred to as a Genetic Algorithm (GA).

The first part of this section will give an overview of heuristics including some examples on related problems. The second part will cover the background on genetic algorithms as well as some issues that should be taken into account when they are applied to real problems.

3.2 Traditional Approaches

To simplify the discussion it might be best to think of the traditional custom made heuristic approach as a "solve as you go along" philosophy. That is, the heuristic algorithm attempts to build the solution incrementally and, when a trouble situation arises, fixes it at that point or goes back to a previous point in the algorithm to try a different route.

Any good textbook on algorithm design and analysis will give several approaches for algorithms including some that give an optimal solution. In these sources traditional heuristics try to take the *best deal* or *greedy* approach to solving problems. Heuristics like this are very common as they are the easiest to implement although they have a disadvantage in which the choices made in the algorithm are based on local optimality. Unfortunately, this means they are not *guaranteed* to generate an optimal or even a good solution in some cases[1] [CLR90]. Some books, like [PVTF92], even have ready to type in code for some heuristics. In general a greedy algorithm is an algorithm that tries to construct the solution in stages where the best available choice at the time is made which locally optimizes the result [Cam94, CLR90].

The Graph Colouring Problem (GCP) will be examined first. Some heuristics that can be used to solve it will be examined as well as a backtracking approach that can find exact (optimal) solutions. The graph colouring problem does have a direct link to scheduling problems as will be described in more detail shortly. The GCP is also known to be NP-Complete [Tuc84]. Before graph colouring is examined a, brief primer on graph theory is required.

3.2.1 Graph Theory

A graph $G = (V, E)$ consists of a finite[2] set of V *vertices* and a set of E *edges*. An example of a graph can be found in Figure 3.1. This graph G has an vertex set

$$V = \{a, b, c, d, e, f\}$$

and also an edge set

$$E = \{(a, b), (a, c), (a, d), (b, d), (c, d), (c, e), (d, f), (e, f)\}.$$

A graph that contains vertices which have edges to themselves are said to contain *loops*. In the context of this work only loopless graphs are used. If two vertices are

[1] With some problems it is possible to prove that a greedy approach will produce optimal results although the problem has to have various properties for this to happen. Scheduling problems rarely have this *greedy choice* property [CLR90].

[2] There is a whole branch of graph theory that uses infinite vertex sets and edge sets but in the context of this work it will not be considered.

joined by an edge they are said to be *adjacent*. A graph that has n mutually adjacent vertices is said to be a *complete graph* on n vertices; this is usually denoted K_n. A *clique* is a subset of vertices in a graph such that every pair in this set is joined by an edge (so the induced subgraph is complete). The *degree* or *valency* of a vertex v in graph G is defined as the number of edges that contain v [Tuc84, Cam94].

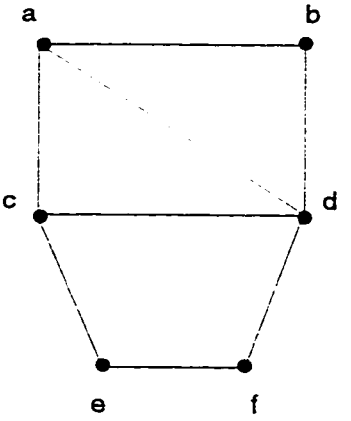


Figure 3.1: Example of Graph G

An edge in a graph can have several properties. An example of this might be *direction* if the graph represented, say, a network of pipeline flows or the flights in and out of an airport. The type of edges that will be used here are *undirected*; the edges are 2-subsets of vertices rather than ordered pairs (which imply direction) [Cam94]. Values can also be associated with edges to denote a cost or penalty. An example of this would be the length of road that an edge in a highway network graph might represent. Another example would be a value representing the number of students enrolled in two courses simultaneously where a vertex represents a course and an edge is present between two vertices if they have at least one student in common.

3.2.2 Graph Colouring Problem

The task of *graph colouring* is to assign colours to each vertex in a graph such that no two adjacent vertices have the same colour. For example, the graph in Figure 3.2 is coloured with the colour values represented in parentheses beside each vertex.

The graph in Figure 3.2 is said to have a 4-colouring. In general a graph is said

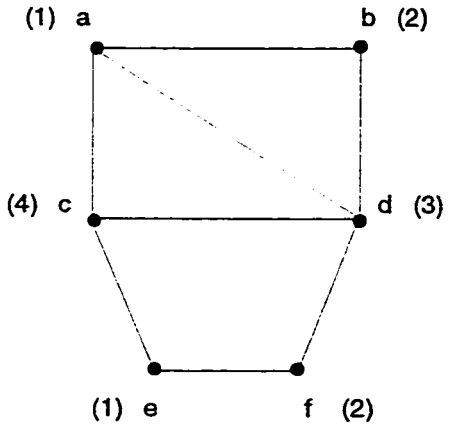


Figure 3.2: Example of a Graph Colouring

to have a k-colouring where k is the number of colours that are used to legally colour the graph. The smallest number of colours needed to colour a graph is said to be the graph's *chromatic number* and is represented by the greek letter χ. This is the minimum number of colours that can be used to colour a graph. For example, with the graph in Figure 3.2, if vertex c is recoloured with colour 2 then 3 is the minimum number of colours that can be used for the colouring of this graph. Since this graph also contains a K_3 clique this shows that the lower bound for the colouring of this graph[3] is three. From this information it can be concluded that $\chi(G) = 3$ for this graph. The graph colouring problem is to colour the graph using the smallest possible number of colours[4].

This problem can be directly applied to examination timetabling if we consider each vertex to be a *course* and an *edge* to exist between two courses if they have at least one student in common. If the vertex colours are considered to be exam periods this corresponds exactly to an examination schedule. This ensures that no student will have to write two exams at one time. The GCP also applies to course scheduling problems. If each vertex is considered to be a course and edges link courses that are not to be scheduled at the same time then a graph colouring (where the colours are time periods) approach can produce a timetable.

A very useful paper on the colouring of graphs by [Man81] provides many easy

[3]Since any clique of size k has $\chi(K_n) = n$

[4]Actually, there are 2 graph colouring problems (at least). The other one is to decide whether or not for a given k, a k-colouring exists.

to implement heuristics as well as a fairly detailed comparison of them. The four heuristics he mentions are: random, largest first, Almost Maximal Independent Sets (AMIS), and colour degree. They are defined as follows given a set V of vertices $v_1, \ldots, v_n$.

1. **[Random]** The random method simply colours the vertices $v_1, \ldots, v_n$ in the order presented, using the smallest permissible colour for each vertex.

2. **[Largest first]** This method sorts the vertices into descending order of their degrees and then applies *random* to it.

3. **[AMIS]** *Almost Maximal Independent Sets.* Let $C = \{$ vertices coloured with current colour $\}$, $N = \{$ vertices adjacent to the set C $\}$. AMIS colours with one colour at a time, selecting for colouring a vertex of minimum degree in $< V \setminus C \setminus N >$, the subgraph induced by $V \setminus C \setminus N$. When $V \setminus C \setminus N$ is empty, the process is repeated with the uncoloured vertices in V using a new colour. This method is actually the worst of the four.

4. **[Colour degree]** This approach defines the *colour degree* of a vertex v to be the number of colours used on the vertices adjacent to v. Colour degree repeatedly colours the uncoloured vertex that has the largest colour-degree with the smallest available colour to it. If two vertices have the same colour-degree then the one of higher degree is done first.

The ranking of these four heuristics, in the context of the quality of the colouring produced, is: colour degree, largest first, random, and AMIS.

Upon inspection by the reader it should be noted that the approach of algorithms two and four is to try to identify the vertices that are the *most difficult* to colour and colour those vertices first. All of the algorithms outlined above are also so-called *one-pass* or *on-line* algorithms in that no vertex is coloured more than once. No effort is made to go back and undo what may have been a previously poor choice. These *greedy algorithms* (algorithms two and four) always choose a vertex that appears to be the best at that point [CLR90]. Unfortunately, the end result is that there is no guarantee that the best result, or even a good one, will be generated from the application of these algorithms.

In past work, involving the University of Calgary final examination timetabler, the colour-degree method was first applied to the graph representing the courses and students. This is the first step in generating an examination schedule. After that point, where a colour degree schedule has been generated, an *exact* colouring algorithm is applied to it. The algorithm applied is a slight modification of the one found in [Pee83]. If the colour-degree algorithm coloured the schedule with k colours then the exact colouring algorithm tries to find a $k-1$ colouring, then a $k-2$ colouring, until either an exact colouring is found (the chromatic number of the graph has been found) or a pre-fixed time limit expires (the problem is NP-complete after all, it may *never* finish in our lifetime!). With the final examination timetabler, the actual chromatic number of the graph representing the exam schedule has often been found which is a very encouraging result [Lay93].

The exact algorithm is an example of an application of *backtracking*. The modified version of the algorithm found in [Pee83] essentially uses a colour degree algorithm to select the next node to colour but one of two things can happen at each node (where it is assumed that the algorithm is trying to find a colouring with fewer than k colours).

1. The node can be assigned a colour $< k$. Choose the next node to colour.

2. The node *cannot* be assigned a colour $< k$. That is, the domain of the current set of colours available to be assigned to this node is empty. Backtrack to a previous node that was coloured and assign it a different colour.

The algorithm to be outlined is from [Pee83], who originally took this *exact* algorithm for the GCP from [Bre79]; the reason for including an outline here is that the heuristics for the MTTP to be described later bear some similarities to this algorithm. The algorithm contained in [Bre79] contained two errors, one in the removal of labels and one dealing with a rule for restarting the algorithm if a new solution has been found. The correct version found in [Pee83] will be used. This algorithm will be referred to as the Brelaz algorithm for simplicity. First the corrected Brelaz algorithm for colouring graphs will be given. Let n be the number of vertices in a graph. Let q be the number of colours used by Colour Degree (or some other heuristic). If a node is labelled at any time in the algorithm, it is simply a way of marking a node to

which backtracking is possible and potentially useful. The procedure to label vertices that is called by the algorithm is given below in Figure 3.3. Let w be the size of a clique placed as the first nodes in the graph and let $\{v_1, \ldots, v_w\}$ be the clique vertices. The reason for wanting to place a clique as the first nodes to be coloured using this algorithm is because if at any time the algorithm backtracks to a node in the clique, the chromatic number (χ) of the graph has been found since a clique of size j cannot be given fewer than j colours, thus modifying the colouring of the clique vertices will have no effect on the number of colours used. This also means that the algorithm can finish early because this can potentially cut down on the number of colourings tried. The algorithm is in Figure 3.4.

```
procedure Label(vertex v)
    begin
       for each colour adjacent to v
       begin
          Label a vertex, u, if these conditions hold:
          1.  Smaller rank than rank of v.
          2.  u is adjacent to v.
          3.  u has minimal rank among all the vertices
              of their colour which are adjacent to v.
       end
    end
```

Figure 3.3: Labelling Procedure For Brelaz's Algorithm

One of the most interesting features of the Brelaz exact graph colouring algorithm is the criteria it uses for backtracking. Clearly the only time backtracking has to be considered is in step 4 when the colour that is to be assigned is the same as or greater than q, which is the colour that is to be improved upon. The algorithm, at this point, has to decide how far to backtrack based on the closest vertex which is labelled. The algorithm then starts colouring again from this vertex by assigning it a different colour from what it previously had. The work in this step is carried out in the Label function presented in Figure 3.3.

Suppose some vertex, say v_i, cannot be coloured because, for each possible colour, at least one neighbor of v_i has already been assigned that colour. Hence backtracking is to be applied from v_i. Now suppose that, for some colour c, at least two neighbors

1. Colour the clique vertices. That is, assign the first w vertices the colours 1 through w. Set q to a value representing the maximum number of colours that are to be used to start. q can initially be calculated from a heuristic such as *colour degree*.
2. $i = w + 1$, Label all clique vertices.
3. Find lowest possible colour j for v_i of those colours that are available.
4. If $j < q$ Goto Step 10.
5. Call procedure Label(v_i).
6. If v_i is labeled, unlabel it.
7. $i = i - 1$.
8. if $i \leq w$ then the most recently saved colouring is optimum, Exit.
9. if v_i is labeled then mark the current colour of v_i as tried by v_i so it will not be tried again on this vertex unless the algorithm backtracks past it and Goto Step 3 else Goto Step 7.
10. Assign this colour j to v_i.
11. $i = i + 1$.
12. if $i < n$ Goto Step 3.
13. Save this new colouring. $q = q - 1$. Unmark all colours on all vertices. Unlabel all vertices. Goto Step 1.

Figure 3.4: Brelaz's Modified Algorithm

of v_i, say v_j and v_k, have been assigned c, and suppose also that v_k occurs later than v_j in the sequence in which the vertices are selected for colouring; in other words v_j has a lower ranking than v_k in this sequence. In that case, there is no point in backtracking to v_k, because assigning a different colour to v_k would still not make colour c available to v_i: the earlier assignment of c to v_j still prevents that. Therefore, for each possible colour c, only the lowest-ranking neighbor of v_i assigned colour c is labelled as being a suitable target for backtracking. Then, among all the vertices so labelled, the one of the highest rank, say v_t, is chosen as the actual target to backtrack to, since there is at least a possibility that by assigning a different colour to it, a suitable colour for v_i will become available. The colour currently assigned to v_t is marked off as invalid,

so it will not be selected again (Unless, of course, the algorithm backtracks to a rank below this vertex sometime in the future). This ensures that new colourings will always be tried and new areas of the solution space are explored. Since this problem is NP-Complete there is no guarantee that it will finish in any reasonable length of time. To counter this a time limit on the runtime of the algorithm is used[5]. As an aside, a modified version of this algorithm is currently used in the generation of examination timetables at the University of Calgary [Lay93, CL95b].

3.3 Genetic Algorithms (GA)

The approach of *genetic algorithms* (GA) involves intelligently exploiting a random search by using an analogy to a natural process [Mic92, Ree93]. GA's try to mimic the idea of survival of the "fittest". In nature, usually the stronger creatures tend to survive while the weaker ones tend to perish. This also works for some of their inherited traits. The traits of the surviving creatures are combined and are carried on from generation to generation while, in general, getting better and stronger. The interest in this approach began in the 70's when Holland first published his "Adaptation in Natural and Artificial Systems" paper [Hol75]. A good way to see the intuition and where the idea of genetic algorithms came from can be found in any junior biology textbook. Take an excerpt from [Cam93] for example:

> "... This continuity of traits from one generation to the next is called heredity. Along with inherited similarity, there is also variation: Offspring exhibit individuality, differing somewhat in appearance from parents and siblings. These observations have been exploited for the thousands of years that people have bred plants and animals."

If dealing with genetics and interbreeding the better plants and animals together produces better strains of these organisms, can this idea be extended to apply to the realm of heuristics and Operations Research (OR) in computer science?

Recall from the previous discussions that the more traditional heuristic methods fail to search out a large area of the solution space. They tend to get bogged down in

[5]Which implies that the best colouring is not necessarily found.

a local minimum. Genetic algorithms seem to have found a way around this problem such that more of the solution space gets examined[6].

First, an explanation will be given to define and explain how genetic algorithms appear to work. This explanation is taken mostly from [Ree93, SP94, PD95, Cam93, Mic92, Dav91] as they give an excellent review of both the theory of how genetic algorithms work[7] as well as the practical aspects of how they may be applied to different problems. With genetic algorithms, the obvious analogy from nature being used is the idea of the survival of the *fittest*. This corresponds to the solution space as mostly promising and good solutions are to survive while the weaker and poorer solutions will tend to disappear. This can be thought of as the *natural selection* of the algorithm towards the population of solutions that is being cultivated. Each solution has to contain various properties of the problem being solved (for example, order of cities in the traveling salesman problem or perhaps the bits making up different values of x in the function $g(x)$ that is to be minimized or maximized), and these properties must be encoded into the solution. Usually the solution is encoded as a *string* (also referred to as a *chromosome* or a *solution*) and this string is made up of variables referred to as *genes* [Ree93]. These chromosomes can be thought of as different *solutions* to the problem. This corresponds nicely to the situation in biology as chromosomes, in the biological context, consist of a long string of genes that each contain units of hereditary information [Cam93]. The traditional approach to encoding chromosomes is to represent them as a string of binary digits. From [SP94] an outline of a simple genetic algorithm is given in Figure 3.5.

The first step of initializing the population usually involves generating a set of chromosomes or solutions (hence known as the *population* P of size N, where $P = \{p_1, \ldots, p_n\}$ and $|P| = N$). The next step involves the evaluation of the population. With each problem that a GA is trying to solve (to which the chromosomes represent solutions) a *fitness function* must be defined that evaluates the solutions individually, say $f(p_i)$. This function calculates a *fitness* evaluation for each chromosome that represents how good the solution is and provides a metric by which solutions can be compared with one another. For example with the travelling salesman problem it

[6]This, of course, does not *guarantee* that the best solution will be found.

[7]Although it is interesting to note that [PD95] seems to disagree with Holland's theoretical conclusions as well as the point of view of [Ree93, SP94] who tend to echo Holland's views.

1. Initialize population.
2. Evaluate population. This involves applying an evaluation function to each member of the population thereby rating each individual on a "fitness scale".
3. While (termination criteria not reached) do
   (a) Select solutions for next population.
   (b) Perform genetic operators such as:
      - Crossover (This is the mating of pairs of chromosomes).
      - Mutation.
   (c) Evaluate population.

Figure 3.5: Simple Genetic Algorithm

would be the length of the tour given by the solution represented by p_i or with the minimization of $g(x)$ it would be the value of $g(p_i)$.

3.3.1 Chromosome Representation

An important issue with GA's is how the chromosomes are represented for the problem being approached. Consider the following, admittedly a rather artificial example. Suppose a function, $f(x)$, is given and GA's are to be used to find the value of an integer x in the range $[0, 255]$ such that $f(x)$ is maximized[8]. An obvious representation for the chromosomes in this case would be to use eight bit integers as chromosomes in the population where each bit would be a gene in the chromosome. As another example recall the graph colouring problem described earlier in this chapter. Offhand it does not appear to have an obvious chromosome representation for GA's to be applied. Looking at the various heuristics outlined for GC it can be seen that if the random algorithm is used for the evaluation function then a graph colouring problem can be represented in a chromosome that is a permutation of vertices in the graph. In this case each vertex would be a gene. This intuitively should be a good representation; after all, the other one-pass heuristics outlined before only rearranged

[8]Obviously the easiest thing to do would be to just try all values of x but in practice the domain of the function is large enough to make this impractical. For example, maximizing a function in 3 or 4 variables over the reals.

the order in which the vertices were coloured. In this re-arranging process it is often the case that some good solutions are ruled out. With this permutation representation the vertex re-arranging is carried out through the GA process.

An important issue arises from this. Once a chromosome representation and a fitness function have been decided upon it must be shown that an optimum solution can be represented with them. In the case of the GC problem the question can be stated more formally:

> Does there exist a permutation of the vertices of a graph G such that the random algorithm can be applied to it such that an optimal colouring can be found?

The answer to the question in this case is yes. Since graph G clearly has an optimum colouring then such a permutation can be constructed. For example, ordering the vertices by the value of their colour[9] will produce a permutation that the random algorithm will evaluate to an optimal colouring. This illustrates the point that careful consideration must be given to how a GA represents a solution in a chromosome. In fact, it may be more desirable that it can be demonstrated that *all* optimal solutions can be represented with the chromosome representation chosen or, at the very least, a substantial proportion of them. Similar care must be taken in the evaluation function.

Chromosome representation also has another facet to be explored. In the above graph colouring example *every* chromosome represents a legal solution (as the fitness function is essentially a graph colouring heuristic). In this case the genetic operators used would move from feasible solution to feasible solution. There are some cases where that cannot be the case. Finding a feasible solution with some problems is difficult and the initial population of solutions in the GA are mostly or entirely made up of illegal solutions that are hoped to improve to feasible solutions over time. The MTTP is a prime example of this as will be seen in Chapter 5 when the Genetic Algorithm used for this problem is described.

3.3.2 Chromosome Selection

Chromosome selection is the procedure in which a GA selects which chromosomes

[9] Using the optimal colouring of course.

are to mate to produce new offspring to put back into the chromosome population. Assume that the population has been evaluated with the *fitness function*, $f(x)$, the focus of this discussion will be how the next generation of the population arises from the old one via *chromosome selection* and the application of *genetic operators* to the chromosomes. Genetic operators, or the *mating* of chromosomes from which new solutions are generated, will be dealt with in the next section.

The idea behind genetic algorithms is to mate various solutions in the hope that they will yield better solutions (they both may contain properties that *make* them good solutions; it is these good properties that are desirable to try to combine in some fashion). The process of selection is the part that tries to model nature's "survival of the fittest" theme.

Chromosomes that are are to be mated are selected based on their fitness ratings. The chromosomes that have the better fitness ratings should be more likely to be chosen to reproduce. The reasoning for this should be obvious as the more fit chromosomes are likely to have properties that are desirable to transfer to further solutions. The less fit chromosomes also have a chance, although somewhat smaller, of being picked for reproduction. Occasionally selecting some of the poorer solutions for reproduction helps to keep some genetic diversity in the population. This, in turn, can lead the solution away from a local optima[10]. Many methods have been described in the literature but only a few will be elaborated in detail here. The method outlined by [Ree93] involves selecting one parent based on its *fitness rating* and then mating it with another chromosome at random. The resulting offspring then replaces a random member of the population P. One method of selecting the parent chromosomes for reproduction given in [Dav91, Mic92] is called the *roulette wheel parent selection*. The idea of this method is to make the chance of a chromosome being selected equal to the proportion of the chromosome's fitness relative to the total fitness of the population, F_t. This is conceptually like having a roulette wheel with each chromosome's, p_i, slice of the "pie" equal to $\frac{f(p_i)}{F_t}$. The algorithm given for parent selection, as given in [Dav91], can be found in Figure 3.6.

This method has problems that may not be initially apparent. While it may

[10]It is the case, of course, that if there were no local optima, one could simply mate the very best chromosomes and get a perfect or excellent result. In this case the algorithm would, in effect, be similar to a greedy algorithm.

1. Sum the fitness of all the population members, $F_t = \sum_{i=1}^{n} f(p_i)$
2. Generate a random number $r \in [0, F_t]$.
3. Return the first population member whose fitness, added to the fitness of the preceding population members, is greater than or equal to r.

Figure 3.6: Roulette Wheel Parent Selection Algorithm

seem like a good idea to select the parents of the next generation based only on their fitness there can arise the case where the population may have individuals whose fitness values are very high compared to the rest and, therefore, have a large number of offspring which may prevent other individuals from contributing and leading to getting stuck in a local minimum. These individuals are sometimes referred to as *super individuals*. They are super with respect to others in the current population, but can represent a solution from a local minimum in the search space. Other selection mechanisms based on *rank* are sometimes used as an alternative to roulette wheel selection. This is the case where the population is ordered by fitness and the higher ranked solutions have a better chance of being picked than lower ranked individuals but the chances remain the same from generation to generation rather than fluctuate with the fitness of the various population members [Mic92]. It is important to ensure that the lesser chromosomes also have a chance to reproduce, in order to help the solutions traverse more of the solution space. This too goes back to nature as the less strong and fit creatures do sometimes manage to survive and contribute to the next generation. The idea behind this model is to make it *less likely* to get stuck in a local optimum.

3.3.3 Genetic Operators

Once the parents for the next generation have been selected, the application of various *genetic operators* is carried out in order to create a new generation of chromosomes. The two basic genetic operators that are predominant in the GA literature are known as *crossover* and *mutation*.

The function of crossover is to combine two parent chromosomes and create two offspring chromosomes that have characteristics of both parent chromosomes. This is

the point where it is hoped that the good properties of solutions are copied over into the next generation (via this crossover operation) and better solutions are created. The goal of the crossover operator is to try to combine properties of current solutions and, to some degree, try to examine more of the search space with the offspring created.

The goal of mutation is to try to ensure that there is some variation in the solution space that is to be examined and to ensure that no point in the search space has a zero probability of being visited [BBM93a]. Mutation, as mentioned earlier, simply replaces a value of a gene with a random one; mutation is, thus, a unary operator.

An important point to notice is that the representation of the chromosomes together with their associated genetic operators (crossover, mutation and possibly others) *must* be able to produce all feasible solutions. In the travelling salesman problem (TSP), for example, the chromosomes and genetic operators should be able to generate *all permutations* of cities.

Some examples of crossover operators will be presented next.

The classic crossover operator is known as a *one-point crossover*. This is simply choosing (at random) a position at which to "cut" the two chromosomes, followed by creating two new chromosomes by picking the first part from one parent's chromosome and the second part from the other parent's. An example will help in visualizing this operation. Consider the two chromosomes:

$$p_1 = 01101010$$

$$p_2 = 10101101$$

If these two chromosomes were to be mated a random number would be generated between one and the length of the chromosome minus one; take three for example. So the one point crossover at position three would look like the example in Figure 3.7 where o_1 and o_2 are the offspring of p_1 and p_2 and "|" represents the crossover point. Just because two parents are mated does not mean they will necessarily produce offspring. There is usually a probability associated with each mating pair known as the *crossover rate* or *crossover probability* which is the probability that offspring will be produced, otherwise the parents are left intact. The *mutation* operation is also

$$p_1 = 011|01010 \qquad o_1 = 01101101$$
$$\rightarrow$$
$$p_2 = 101|01101 \qquad o_2 = 10101010$$

Figure 3.7: Example of 1-Point Crossover

carried out after the mating has finished (whether offspring was produced or not). Each *gene* in each surviving chromosome has some probability of mutation (known as the *mutation rate* or *mutation probability*) where that location will be altered (usually randomly). For example if mutation was to be applied to the third gene of o_1 in the above example the "1" in that location could be switched to a "0". Many authors have tried a wide variety of approaches to crossover and mutation.

As examples of the many variations of crossover that have been suggested, consider the following. More than one point can be used in crossover. *Two point* crossover can be used. Two points are selected instead of the one, as above, and the partitioned portions of the chromosomes are swapped similar to what happens in one point crossover. Another method of crossover that can be useful is known as *uniform crossover* and the idea behind this method is to determine randomly which parent provides the gene for each child. This can be represented in a template that is the same length as the chromosome [Fan92, Dav91, Sys89]. Each position in the template is given a "1" or a "0" entry and determines which parent will contribute that gene position to the offspring (while the second offspring gets the other parents' gene).

An example of uniform crossover can be found in Figure 3.8.

$$\begin{aligned} p_1 &= 10110110 \\ p_2 &= 00101011 \\ \text{Template} &= 10011011 \\ o_1 &= 10110010 \\ o_2 &= 00101111 \end{aligned}$$

Figure 3.8: Example of Uniform Crossover

These crossover operators are, obviously, designed for chromosomes with a binary alphabet. For chromosomes with alphabets of a higher cardinality or chromosomes that have some restrictions placed upon them (like a permutation), these crossover operators may not be suitable. Many authors have experimented with a wide variety of chromosome representations and operators for a wide variety of real world problems [Mic92, Dav91, Fan92]. As will be seen later, the simple operators outlined above often turn out to be useful with more complex chromosome representations.

3.3.4 Evaluation Function

The evaluation function, $f(x)$ – and its interplay with the reproductive stage – is the driving force behind the GA. It supplies the *selective pressure* that decides which chromosomes do well and which die off. If the evaluation function gives too much emphasis towards good chromosomes then some super individuals may take over and end up reproducing more than they should thus thinning out the gene pool with respect to other not so fit individuals. This could cause the algorithm to settle on a local minimum. On the other hand if weak selective pressure is being used then this can also make the search ineffective or even make it degenerate into a random search altogether as the chromosomes may well be rated virtually equal throughout the life of the program. It is the evaluation function that the GA is trying to optimize so it is important that the quality of the solution is reflected in its value.

3.3.5 Genetic Algorithm Schema Theory

Many algorithms and heuristic methods have a theoretical foundation that makes them seem promising to employ. The area of genetic algorithms is no exception and any discussion of it would be incomplete without a theoretical foundation being presented to help give the reader some intuition as to why this may be a promising method to try.

The theory used to account for some of the success of genetic algorithms is known as the *schema theory*. [SP94] describes a schema as a *similarity template* that describes a subset of strings with similarities in some of their positions[11]. For example

[11]The schema theory assumes that the alphabet of the chromosomes is a binary one.

the *schema* 001#0# represents the following set of strings:

$$\{001000, 001001, 001100, 001101\}$$

The "#" characters in the schema can be thought of as wild cards. Each string in the population that is represented by a schema is called an *instance* of that schema. The number of fixed positions contained in a schema is referred to as its *order*. The distance between the outermost fixed positions in a schema is known as its *defining length*. For example, consider the schema $\mathbf{h} = 001\#0\#$; the order of $\mathbf{h}$ is $o(\mathbf{h}) = 4$ and the defining length would be $\delta(\mathbf{h}) = 4$.

If the example is examined further it will be noticed that there are 4 fixed positions in the schema and it can be seen that there must be $2^4 - 1$ other schemas with the same fixed positions (consider all other combinations of 1 and 0). In general, in a schema with k fixed positions, there are exactly 2^k schemas that will have the same k fixed positions. For each string of length l, there are 2^l matching schemas in total since at every point it can take a normal value or the wild card "#". Provided, of course, that reasonable choices have been made for both the fitness function and chromosome evaluation, it should be the case that, as evolution continues, the number of some schemas in the population will increase while other schemas will start to become less predominant. This is what is known as the *schema competition*. A theorem known as the *schema theorem* helps explain why good schemas can be expected to have a higher chance of survival than than poor ones do. This, in turn, implies that it is more likely that good schemas will flourish as opposed to schemas found in less fit solutions.

The idea behind the *schema theorem* is to show that the schemas that represent the *stronger* chromosomes, i. e. the ones corresponding to the higher values of the evaluation function, in the population should have their representation *increased* in further generations (since the idea is for the strongest to survive and flourish and, therefore, be more numerous in the population).

The basic idea of the schema theorem is that, with the effects of the genetic operators (crossover and mutation), it should be the case that the instances of various schemas increase or decrease with relation to the average fitness value of the population. It is worthwhile to note, at this point, that there is a connection between the

fitness function and the chromosome. Clearly the fitness function should give varying results that reflect the general quality of each chromosome for if the fitness function gives fairly "flat" results then not many schemas will evolve at all. This is due to the fact the chromosomes all have similar fitnesses and, therefore, it is difficult for the GA to put pressure on any particular solution as they appear to be very similar in quality.

Taken from [Ree93][12] the schema theorem is as follows (The reader is referred to [Ree93] for a proof):

Theorem 1 *Using a reproductive plan similar to the one outlined in [Ree93] in which the probabilities of crossover and mutation are Pr_c and Pr_m, and schema* $\mathbf{h}$ *of order* $o(\mathbf{h})$ *and defining length* $\delta(\mathbf{h})$ *and length* l *with a fitness ratio*[13] *of* $\frac{f(\mathbf{h},t)}{\overline{f}(t)}$ *at time* t *, then the expected number of representatives of schema* $\mathbf{h}$ *at time* $t+1$ *is given by:*

$$N(\mathbf{h}, t+1) \geq N(\mathbf{h}, t)\frac{f(\mathbf{h}, t)}{\overline{f}(t)}\left[1 - Pr_c\frac{\delta(\mathbf{h})}{l-1} - Pr_m o(\mathbf{h})\right]$$

In the above theorem the term $Pr_c\frac{\delta(\mathbf{h})}{l-1}$ represents the probability a particular schema will be destroyed by crossover. The term $Pr_m o(\mathbf{h})$ represents the probability that a schema will be destroyed by the mutation operation. This theorem demonstrates that if the representatives of a particular schema $\mathbf{h}$ are *better than* average[14], then in the next iteration they should increase in number. This leads us to another concept known as the *building block principle*. For schemas to be effective and to have a good chance of surviving from generation to generation they should be short low order schemas. These schemas are thought of as *building blocks* and the idea that they are good for building better solutions by combining these "building blocks" is known as the building block principle or building block hypothesis. The reason for the shorter schema length being a desirable property is due to the fact there is less chance of *disruption* during the crossover process. If the crossover point fell between

[12] Actually, the notation for the equation below is borrowed from [Mic92] since I found it much easier to follow (and appears to be the more common form of representation in the literature) than the notation used in [Ree93].

[13] With $f(\mathbf{h}, t)$ to be the *average* fitness of all the instances of schema $\mathbf{h}$ in the population at time t. The representation of the *average fitness* of the population at time t is $\overline{f}(t)$.

[14] That is $\frac{f(\mathbf{h},t)}{\overline{f}(t)} > 1$.

the fixed positions of the schema or mutation was carried out on a fixed position in a good schema the instance of this particular schema would be destroyed. This is represented in the above equations by the terms representing the probability that a schema will be destroyed by crossover or mutation.

The schema theorem makes the assumption that the chromosomes being used are strings of binary digits. Unfortunately, there is no accepted "general theory" that explains why GA's work the way they do [BBM93a]. In real-life problems it is often not possible to represent the solutions (chromosomes) as strings of bits. An example would be the graph colouring problem as outlined earlier. As will be seen, the chromosome representation of the MTT problem will not be a bit-oriented one. A similar analysis should in principle be possible, with a similar result, that is to say that representatives of schemas that are better than average should increase in number over time. Most GA research, however, concentrates on finding empirical rules for GA's to give good results [BBM93a].

3.3.6 Variations

One of the more interesting features of genetic algorithms is that there are many variations of the ideas outlined above. The basic genetic algorithm outlined is sometimes referred to as a "SGA" (Simple Genetic Algorithm). There are many variations on methods to select parents for offspring, and many different ways of performing crossover and mutation. This section gives the reader some background on some heuristic methods as well as an in depth discussion of GA's and their components as well as their interesting link to biological methods. The next two chapters will explore how heuristics and GA's are applied to the MTT problem.

Chapter 4

Application of Heuristic Method

The Master Timetable Problem was first examined at the University of Calgary in the summer of 1994 as a summer project. The student working on it designed some of the data structures that are still used in the method developed here but, unfortunately, did not proceed far enough to consistently produce viable timetables.

This chapter will describe a heuristic method that has been developed that will produce good timetables (comparable or better than the timetable in actual use) for some of the most difficult departmental data at the University. Before the method itself can be discussed it would be worthwhile to first examine the data being used as well as what makes it difficult to find a solution for this set of data.

4.1 Data Used

Two datasets for the master timetable problem are considered to evaluate the methods employed. The first set of data is from the Department of Computer Science at the University of Calgary. This timetable is not very restrictive (the most restrictive term was partially outlined earlier in Figure 2.1 regarding second year computer science) compared to the second set of data. As a result it was used more for the early testing of the methods used and not focused on as the main test dataset.

The second set of data used for testing purposes is from the Departments of Biology and Chemistry at the University of Calgary. Obtaining a feasible schedule for this set of data proved to be very challenging. In addition, in most of the literature

that was considered involving the master timetable problem (or variations on this theme with regard to different institutions' rules and constraints), it would appear that the courses to be scheduled had *fewer* restrictions than what was found with this second set of data in the sense of how "tight" the data is. Also, the data size used by virtually all of the other researchers' work examined appeared to be considerably *smaller* in size; thereby, in some sense, making it not as difficult as the Biology and Chemistry data [RCF94b, Ran95, Ric95]. To be fair, many of these cases were class-based as opposed to student-based which makes comparison between the two methods difficult due to the intrinsic differences between the two methods. Other institutions also have different rules they apply to schedule generation and different conflict criteria than that used at the University of Calgary. This may be why a general solution has not been found to the school timetabling problem as every school or university seems to have their own rules making a general solution difficult to visualize.

4.1.1 Data Format

The data that contains the timeslot and room categories[1] as well as information on the calendar requirements and course sections to be scheduled are stored in four different files. They are:

1. The **combo.dat** file[2]. This file contains the information about the sections that are to be scheduled as well as their room and timeslot categories

2. The **rooms.dat** file. This file contains data representing all of the rooms and room categories that are to be used.

3. The **ts.dat** file. This file contains data representing all of the timeslots and timeslot categories that are used.

4. The **calendar.dat** file. This file contains information about any restrictions that are to be applied to the timetable. It is information such as what can be

[1] Definitions of room and timeslot categories can be found in Chapter 2.

[2] The name **combo.dat** is what the summer student who first started this project called the file and the courses. His terminology for them was *combos* and some of this terminology has lived on into this later version of the project!

found in Table 2.1 regarding what courses are generally *required* for a certain program in a given term/year.

Partial samples of the data files that were actually used as well as an explanation of the file format can be found in Appendix A.

4.1.2 Data Structures

From the data files outlined in the previous subsection it is necessary to make some general data structures to store this data in order to ensure that the information contained in these files can be looked up both easily and quickly.

The information supplied in the **combo.dat** file is stored in a simple array. There is one array element for each section of a particular course lecture/laboratory/tutorial.

The rooms from the **rooms.dat** file are stored in an array that contains the information about each room. An array of room *categories* is created also containing a list of the rooms that are contained within each room category.

The timeslots from the **ts.dat** file required careful consideration as to how they would be represented. Some method had to be found to store the timeslots so it is easy to compare which timeslots overlapped with one another. Timeslots are stored in a *weekmap* data structure which is simply a 7×32 bitmatrix. This bitmatrix represents seven days of the week in half hour blocks of time from 7:00 in the morning till 10:00 at night. For example, a weekmap representing the timeslot MWF 12:00 50 would look like the structure found in Figure 4.1.

	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Sunday	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
Monday	00	00	00	00	00	11	00	00	00	00	00	00	00	00	00	00
Tuesday	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
Wednesday	00	00	00	00	00	11	00	00	00	00	00	00	00	00	00	00
Thursday	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
Friday	00	00	00	00	00	11	00	00	00	00	00	00	00	00	00	00
Saturday	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 4.1: Example of the Weekmap representing MWF 12-13

The size of this weekmap neatly fits in the possible times and days that a timeslot may cover. Also, since a weekmap is simply an array of seven integers (of 32 bits) it is easy to carry out bitwise operations such as AND/OR/XOR between different weekmaps. For example, if a logical AND operation were carried out between two weekmaps representing timeslots and at least one of the resulting integers was not equal to zero then it can be said that these two timeslots overlap with one another. In fact, an operation is carried out between all of the timeslots as soon as they are read in and converted into weekmaps to create a *Timeslot Conflict Matrix*, T_{cm}, such that:

$$T_{cm}[i,j] = 1 \quad \text{If timeslot } i \text{ and } j \text{ conflict.}$$
$$T_{cm}[i,j] = 0 \quad \text{Otherwise.}$$

This provides a fast and easy way of checking for timeslot conflicts. Another use of the weekmap is to also attach one to each *room* (one for each term to be scheduled) and use it to represent the times the *room is booked*. So, for example, if the above weekmap were for a room then it would be considered to be booked on MWF from 12:00 till 1:00. This turns out to be useful in the heuristic approach to determine which rooms are available for a section at a particular time. Clearly this weekmap structure gives the programmer a very fine tool to change any temporal requirements as seen fit at any time in the program. That is, any special requirements or restrictions on times for rooms (or special timeslots) could be added easily.

As in the case of room categories briefly explained above, an array of *timeslot categories* is created listing the timeslots that are present in each category as they are presented in the **timeslot.dat** file[3].

Another matrix must be constructed that reflects the constraints between courses that should be taken into account. This will be referred to as the *Calendar Conflict Matrix*, C_{cm}. The information to construct this matrix is found in the **calendar.dat** file. The matrix has the properties that for any two courses i and j:

$$C_{cm}[i,j] = 1 \quad \text{If courses } i \text{ and } j \text{ conflict.}$$
$$C_{cm}[i,j] = 0 \quad \text{Otherwise.}$$

[3]Timeslot and room categories were defined in Section 2.1.1.

The entries $C_{cm}[i,i]$ equal one as a course should be considered to conflict with itself as in the case of multiple lecture/laboratory/tutorial sections (in general) they should not be scheduled at the same time. Laboratory and lecture sections of the same course component, for example, should not be scheduled at the same time (although for large multiple laboratory sections it can be permissible to have laboratory sections scheduled in overlapping timeslots). The conflicts found in this matrix represent constraints on the two courses in question in that they should not have overlapping timeslots in the master timetable.

These data structures are straightforward to create using the data in the appropriate files. They also give the programmer the ability to look up useful facts quickly which is important in combinatorial problems as they can sometimes run for quite some time[4].

4.1.3 Possible Difficulties

Before the MTT problem can be tackled it must be known what is inherently "difficult" about making a schedule such as this[5]. If these difficult points can be spotted ahead of time then the heuristic can be modified to deal with them accordingly.

First, it will be worthwhile to define some terms that will be used. The term 'single section' refers to a section that has the property that it is the only one of its instruction type for a particular course, for example, the *only* BIOL 231 lecture or laboratory section. Since there is only one of these sections to be scheduled it is all the more important that it be done right. The term 'multiple section' refers to a section that is one of many of its instruction type for a particular course. For example, a course like BIOL231 will have several laboratory sections to schedule.

The Biology/Chemistry scheduling data has been identified as the more difficult of the two that are used as test data for this project. It is also one of the main motivations for this project to be undertaken. The reason this data is seen to be so difficult to schedule can be found in both the calendar constraints on the courses available as well as the large number of multiple laboratory sections involved. These sections are large in the sense that a Biology or Chemistry lecture section(s) may

[4]For example, the final examination timetabling system can run for well over an hour at times. The use of faster fine-tuned routines can sometimes cut down a running time by a noticeable amount.

[5]Specifically regarding the Biology/Chemistry data in this case.

have upwards of 20 or 30 laboratory sections associated with them. It is (usually) the make up of a course that, if a student is registered in a lecture section that also has lab sections then, the student must also be enrolled in one of the related lab sections. If such a student has to take several Biology or Chemistry courses at the same time that are co-requisites then it can be seen that it becomes a very difficult problem to schedule all of these laboratory sections while still leaving as many lab sections "open" to these students as possible. If these large classes have several lecture sections as well as many lab sections then this problem can become more complex. Generally it is the case that as many lecture and lab sections as possible (if they have a constraint between them) should be kept open for students to take (that is, not scheduled at the same time). This is one of the inherent disadvantages of using a student based schedule as any possible conflicts must be taken into account *before* any student actually tries to register in a certain set of courses.

4.2 Method

The method that is developed here is similar to the one already presented in Chapter 3 regarding the Brelaz graph colouring algorithm. The general approach taken is to consider each section in order and assign a timeslot and room to it such that none of the constraints in the **calendar.dat** file are violated. Unfortunately this problem turns out to be much more complex than the graph colouring problem, especially in the case of generating a MTT for the Biology and Chemistry data for the reasons that are outlined in the previous section.

First, it will be worthwhile to give an outline of the basic algorithm that is used. Next an explanation of how it was developed and where it stands today will be given. The algorithm can be found in Figure 4.2.

The algorithm as presented, obviously, does not take into account all the details that are important in the development of a good heuristic but instead gives an outline of the steps that are needed. The routines will be examined individually in more detail later in this chapter but an explanation of the basic method involved will be presented first.

For each section, 1 through n, an attempt to schedule it is made which involves

```
procedure Schedule() {
   int i = j = 0;
   char Flag[n];
   while (i < n) { /* n is the number of courses */
      if (ScheduleSection(C_i)) { /* Try to Schedule Section */
         BookRoom(C_i);    Flag[i] = FALSE;
         UpdateConflicts(i);    i++;
         /* Updates remaining sections w/r to i */
      } else { /* Else...  Backtrack */
         for (j = 0; j < i; j++)
            if (Conflict(i, j))
             /* Flag possible backtrack destinations */
               Flag[j] = TRUE; else Flag[j] = FALSE;
         for (j = (i - 1); j ≥ 0; j- -) /* Backtrack */
            if (Flag[j]) break;
         if (j == -1) exit(1); /* No Solution */
         ReSet(i, j);  i = j; /* Reset Sections and Unbook Rooms */
      } /* ELSE */
   } /* WHILE */
} /* Schedule() */
```

Figure 4.2: The Basic Heuristic Algorithm

(the while loop):

1. Selecting a valid timeslot for the section.

2. Selecting a room that is vacant at that time.

That is, essentially, what the **ScheduleSection**(C_i) and **BookRoom**(C_i) routines accomplish. If this cannot be done (**ScheduleSection**(C_i) returns false), then the algorithm must *backtrack* to a previous point in the search tree to try another choice that will, it is hoped, prevent that problem from arising again. The **Conflict**(i, j) routine tests for a conflict between the sections i and j and if there is one, section j is flagged as a possible location for the backtracking operation. The **ReSet**(i, j) function simply resets all the bookkeeping that was done between courses i and j. This includes the unbooking of rooms that are used in between courses i and j so they will be available for future scheduling. It also undoes all the timeslot manipulation of the **UpdateConflicts()** routine which will be described in more detail later in this

chapter. Another observation to be made is that the algorithm is unable to find a solution if it backtracks past the first section at any point (where the test between -1 and j is made).

The next part of this discussion will be about the various pieces of the algorithm that are outlined above. The first topic to be discussed will be the preprocessing that must occur before the algorithm is started.

4.2.1 Preprocessing

Obvious work must be carried out in the preprocessing stage of any scheduling system. In this case various checks are made on the integrity of the data supplied by the user as it is typed into various data files manually and can potentially contain mistakes or inconsistencies. For example, each room category consists of a list of rooms that belong to it. Many room categories have intersecting sets of rooms available to them and these rooms are checked against one another to make sure the data is consistent. Information, such as room sizes, is compared to make sure that if a room is listed twice, it has the same capacity in both cases, so if there are any problems, they are dealt with early on in the program.

4.2.2 Section Selection

Some other work has to be considered before the algorithm is started. The order in which the sections are dealt with is one such concern and, at first, it might seem like a good idea to sort the sections in the order of "most difficult" to "least difficult". This way the more difficult to schedule sections (such as laboratory sections with few timeslots available to them) tend to be scheduled earlier than easier to schedule sections such as tutorials which tend to have very loose room restrictions as well as more timeslots available to them. The early versions of the program carried out such a preprocessing step in that it sorted the courses based on the number of timeslots available to them as well as the number of sections of each type of course. That is, courses that have few lecture sections available to them are "harder" to schedule than courses with many lecture sections as the students involved will have less flexibility in lecture time selection (since there are fewer lecture sections to choose from).

In practice, however, it was found that an on the fly method was better suited for selecting the next section to be scheduled. This step is not reflected in Figure 4.2. The idea is similar to the Colour Degree Graph Colouring Algorithm discussed earlier in Chapter 3 [Man81].

Since during the runtime of the algorithm the number of valid timeslots available for each unscheduled section varies dynamically, it is more effective to rate unscheduled courses with some sort of "difficulty" rating metric to determine which section should be scheduled next. Equation 4.1 is such a metric that is used to complete this task.

$$r_j = \frac{NumberOfGoodTimeSlots_j}{NumberOfSections_j} \tag{4.1}$$

Each unscheduled section, j, is given a rating, r_j, that is the ratio of the number of good (legal) timeslots it has (this is dependent on the previously scheduled sections) and the number of sections that are the same course and instruction type that are still to be scheduled[6]. The lower the rating r_j the *higher* a priority it is to be scheduled next. The course section containing the lowest r_j rating will, of course, be the next course section scheduled. This equation will give courses that have many sections of the same instruction type still to be scheduled a higher priority (lower r_j value) than courses that have few sections and many available timeslots.

A useful side effect of this equation is that any section that has *zero* legal timeslots will have a r_j value of 0. This has the useful property that backtracking will be initiated immediately.

This rating system is not perfect and rather *ad hoc* but it works well in practice.

4.2.3 The ScheduleSection() Routine

The **ScheduleSection()** routine attempts to assign a timeslot as well as a room to a course section. It returns a boolean value indicating whether or not it was successful in this allocation.

The selection of the timeslot that is to be chosen out of the list of the available timeslots requires a small heuristic of its own. First, each available timeslot is examined and the number of rooms that are available is counted. The timeslot with the

[6] In the same term of course.

most free rooms available to it is chosen. From the rooms available to this timeslot, the room that is *the least heavily used* is assigned for the current section. This is easy to calculate as it involves quickly counting the number of bits set in the rooms' associated weekmap.

If no suitable timeslot can be found this function returns false so that backtracking can take place.

4.2.4 The UpdateConflicts(i) Routine

When a section has been successfully scheduled the **UpdateConflicts(i)** routine is then called. This routine simply updates the timeslot lists of unscheduled sections if they contain any timeslots that may have a conflict with the section just scheduled (parameter i). Specifically, **UpdateConflicts(i)** scans ahead through these unscheduled sections and any that have timeslots that conflict with *the section just scheduled* have these timeslots removed from their lists of available timeslots. The sections in question must, of course, have some sort of constraint between them to justify this operation (a '1' in the calendar conflict matrix for example). While each list of timeslots is being scanned a test is performed on each timeslot to ensure that at least one room is available to the section in question at that time. If a timeslot is found to have no available rooms into which the unscheduled section being examined can potentially be placed then it too will be removed. A timeslot clearly cannot be used if there is no room available at that time. This updating of unscheduled sections' timeslot lists ensures that when sections are considered in the **ScheduleSection()** function they only have legal timeslots available to them. The **UpdateConflicts(i)** routine looks something like that found in Figure 4.3. The **GetTimeslot(j, k)** routine returns the kth timeslot from section j. **MaxTimeslots(j)** simply returns the number of remaining valid timeslots available to section j. The **NoRooms(t)** function is a boolean test to determine whether or not timeslot t has any available rooms.

The unscheduled section's timeslots that are removed by the **UpdateConflicts()** routine are marked with the section that has forced their removal (this is why parameter i is used in the **TakeOutTimeslot()** routine). If the algorithm were to backtrack past this section that has just been scheduled, the timeslots taken out because of this section should be put back into the pools of available timeslots for the currently

unscheduled sections affected.

Backtracking can also be initiated early if it is found than an unscheduled section has an empty list of available timeslots. Fortunately, as mentioned earlier, this is taken care of with Formula 4.1.

```
procedure UpdateConflicts(i) {
   int j, k, ts1, ts2;
   ts1 = GetTimeslot(i); /* Want timeslot of section i */
   for (j = (i + 1); j < NumberOfSections; j++) {
      for (k = 0; k < MaxTimeslots(j) ; k++) {
         ts2 = GetTimeslot(j, k);
         if (NoRooms(ts2))
            TakeOutTimeslot(j, k, i);
         /* Removes timeslot k from section j */
         else if (Conflict(i, j))
            if (T_cm[ts1, ts2] == 1)
               TakeOutTimeslot(j, k, i);
      } /* FOR */
   } /* FOR */
} /* UpdateConflicts */
```

Figure 4.3: The UpdateConflicts(i) routine.

4.2.5 The BookRoom() Routine

The **BookRoom()** routine is pretty much self-explanatory. It simply marks off the bits in the weekmap for the used room corresponding to the timeslot and the term of the section that is assigned there. The weekmap representation is useful in that it is easy to tell how heavily a room is being used by simply checking how many bits are set in the weekmap. This feature is exploited in the room selection heuristic in the **ScheduleSection()** routine outlined earlier.

4.2.6 The Conflict() Routine

The **Conflict()** routine is the key element in this system as it controls which constraints are honoured and, in the earlier versions, labelled sections that were targets for backtracking. This routine also decides which sections were constrained with one

another in the lookahead procedure such that conflicting timeslots can be removed from the lists of timeslots available to unscheduled sections.

Essentially, the **Conflict()** routine returns a value of true or false as to whether or not there is a constraint between two sections (which are passed as parameters). Care needs to be taken as to what constraints need to be honoured and which ones can be overlooked. This routine was used in two different places in the algorithm originally. It is called from the main **Schedule()** routine (for the purpose of labelling potential courses that could be used for backtracking purposes) and it is also called from the **UpdateConflicts(i)** routine which scans ahead to update the timeslots available to any unscheduled sections thus making sure any yet unscheduled sections have a valid timeslot list. This routine has evolved over the duration of this project rather substantially which will be seen.

First Version

The first version of this routine essentially only tested to see if there was a calendar conflict between two sections.

First, it would make a test to see that the two sections in question were in the same term; if they were in different or non-overlapping terms then the two sections cannot possibly conflict. The next step simply required a lookup in the *Calendar Conflict Matrix*, C_{cm}, as defined earlier in this section. This routine also took into account that if two sections were of the same course type but had different instruction types it should honour the constraint between them (that is, ensure that their timeslots do not overlap). An example of this case would be having a constraint between a BIOLOGY 201 lecture section and a BIOLOGY 201 laboratory section (Since a student registered in the lecture must also be able to register in a lab section).

In practice it was found that this made the problem too constrained as it created *too many* conflicts and as such not enough timeslots were available to accommodate all of the sections. To understand this consider again the example of a junior biology course. Such a course may have two or three lecture sections and 30 (or more) laboratory sections. If every course that had a calendar conflict with this biology course was not allowed to occupy an overlapping timeslot with any of the lecture sections as well as the 30 laboratory sections then it can be quickly seen that it

becomes impossible to generate such a schedule. This is mainly due to the fact that these biology laboratory sections will be running virtually all day every day (bearing in mind each of these laboratory sections are usually three to four hours in length). Clearly the **Conflict()** routine must be modified to reflect this in the sense it should be able to identify the pairs of courses where the calendar conflicts are "less important" than the calendar conflicts for pairs of other courses. Another interesting observation to make about some of these large multiple laboratory sections is that many of them have only *one room* they can be placed in due to specialized equipment that must be present for the students to use. This extra constraint (few rooms available for large multiple laboratory sections) is usually taken into account by the fact these sections are often scheduled earlier due to a tight constraint created by a large number of sections to be scheduled into a small number of timeslots (this is done with Equation 4.1).

Second Version

The second version of the **Conflict()** routine tried to take some of these issues into account. The approach taken was to consider how many sections of a certain course instruction type were to be scheduled and the relationship between these sections and other courses. The troublesome courses were usually the ones that had many laboratory sections to schedule.

One case that was taken into consideration was the situation where a course that has many laboratory sections also has multiple lecture sections. Clearly conflicts between the lecture and laboratory sections can be pretty much ignored so long as all the lecture sections do not conflict with the same laboratory sections. It must be the case that every laboratory section has at least one lecture section of its own course type that does not conflict with it otherwise it will be impossible for a student to enroll both in that laboratory section and a lecture section for that course. In practice when conflicts were ignored between multiple laboratory sections and multiple lecture sections (both of the same course type) it was found this problem did not occur often (if at all). An analogous situation arises for courses with multiple lecture and tutorial sections.

Another observation to make is that since there is a large number of laboratory

sections for various courses it is virtually impossible to avoid using timeslots that overlap with those of courses with which they're not supposed to conflict. Because there are so many laboratory sections of various courses there seems to be little real choice in placing them. The "tightness" of the number of laboratory sections of a particular course with the number of rooms available as well as the number of timeslots often did not leave much (if any) room for manipulation. Courses that have large[7] numbers of laboratory sections had conflicts with them *ignored* altogether as the scheduling for these sections is pretty much forced by the number of timeslot/room combinations compared to the number of sections to be scheduled.

These modifications had the desired effect in that they loosened the conflicts enough to produce timetables but the timetables produced were not strict enough in that too many constraints between courses were not honoured. The resulting timetables were not feasible.

The first and second versions of the **Conflict()** routine were used both for the lookahead (updating of timeslot lists in unscheduled sections) as well as the backtracking (labelling of sections) features in the algorithm. The next version of the **Conflict()** routine to be described is used only for looking ahead. A separate routine is used for deciding where to backtrack.

Third Version

The third, and final, version of the **Conflict()** routine tries to take into account how *important* conflicts are between various courses as well as taking into account other problems large numbers of laboratory sections can cause. All constraints as defined in the **calendar.dat** file are to be taken seriously, of course, but some are more serious than others and the less serious ones can perhaps be ignored in some carefully defined situations.

This version of the conflict routine carries over some similarities from the previous two. The **UpdateConflicts()** routine makes use of the **Conflict()** routine to look ahead to each unscheduled section from the perspective of the just scheduled section and ensure the list of available timeslots for these unscheduled sections is valid. The pseudocode found in Figure 4.4 gives an outline of what happens in this version of

[7] Large in this context generally means 16 laboratory sections or more.

the routine.

```
procedure Conflict(i,j) {
    if (ConflictingTerms(i,j) == FALSE) return(FALSE);
    if(C_cm[i,j] == FALSE) return(FALSE); /* No Conflict */
    Same = IsSameNameNumberTerm(i,j); /* Same course type? */
    if (Same == FALSE) /* Not the same course type? */
        if (IsLab(i) AND (NumOfSections(i) > 4) AND (NumOfSections(j) == 1))
            if (Random() < PM[i,j]) return(TRUE) else return(FALSE);
        if (IsLab(j) AND (NumOfSections(j) > 4) AND (NumOfSections(i) == 1))
            if (Random() < PM[j,i]) return(TRUE) else return(FALSE);
        if (IsLab(i) AND (NumOfSections(i) > 4)
         AND (NumOfSections(j) > 1) AND (IsLecture(j))
            if (Random() < PM[i,j]9/10) return(TRUE) else return(FALSE);
        if (IsLab(j) AND (NumOfSections(j) > 4)
         AND (NumOfSections(i) > 1) AND (IsLecture(i))
            if (Random() < PM[j,i]9/10) return(TRUE) else return(FALSE);
    else /* They are the same course type */
        if ((GetInstType(i) == GetInstType(j)) AND (IsNotLecture(i)))
            return(FALSE);
        else
            return(TRUE);
    if (IsLab(j) AND NumOfSections(j) > 4)
      AND (IsLab(i) OR IsTutorial(i)))
        return(FALSE);
    if (IsLab(i) AND NumOfSections(i) > 4)
      AND (IsLab(j) OR IsTutorial(j)))
        return(FALSE);
    return(TRUE);
} /* Conflict */
```

Figure 4.4: The Conflict(i, j) routine.

At first glance this routine appears to be somewhat complex. The first part of the routine simply checks to see if the two sections being examined by the conflict routine are in conflicting terms. If the sections are not in the same or overlapping terms then clearly they can be ignored.

The next line reads in the value in the calendar conflict matrix corresponding to the two sections (i and j) and if this value is false the **Conflict()** routine returns false. If the courses do not conflict in the calendar conflict matrix then they should not be considered here any further. Next a test is performed to see if the two sections are of the same course component type[8].

[8] For example, the two sections both belong to the CHEM201 course type.

The next test looks at the value of the *Same* variable in order to see if the two sections are of the same course component type. If they are not several groups of tests are then carried out.

The first group of tests performed checks that:

1. Section i is of a laboratory instruction type.
2. Section i has more than four sections to be scheduled.
3. Section j is of a single section instruction type.

If the above tests all hold true then a probabilistic test is performed that decides whether or not the constraint between these two sections will be honoured by returning a value of true or false respectively. This probability is explained in more detail later in this chapter. This is an important step in the conflict routine. It is single sections such as these (section j in the test) that seem to be difficult to schedule with respect to multiple section laboratories. It is desirable to honour all the constraints on the timetable but at the same time it is impossible to do so. The probabilistic test tries to ensure a reasonable proportion of the potential conflicts will be honoured. The identical test is made again swapping i for j.

The next tests done check that:

1. Section i is of a laboratory instruction type.
2. Section i has more than four sections to be scheduled.
3. Section j has at least two sections.
4. Section j is of a lecture instruction type.

Similar to the two previous tests, a probabilistic test is performed that will decide whether this constraint will be honoured or not. In this case the probability used in the probabilistic test is scaled by a value of 0.9. This is due to the fact that constraints between multiple laboratory sections and multiple lecture sections are not as serious as constraints between multiple laboratory sections and single section instruction types. It is also the case if this value is not scaled then the problem becomes too highly constrained as too many constraints will be honoured. In both cases, of course,

some effort must be made in trying to honour some of these constraints otherwise the schedule will be infeasible because of too many laboratory sections conflicting with too many lecture sections and single section instruction type sections.

There is a reason for the value of four being selected as mentioned in the previous paragraph. Through experimentation it was found that the value of four seems to be the dividing point between where constraints have to be carefully dealt with (values up to four) and the case where constraints need not be so strictly dealt with (values of five or more).

It is also worthwhile at this point to highlight a trick that is used with the data to lessen the complexity of the problem. A good example would be a junior biology course. Such a course has, say, four rooms where laboratory sessions can take place (dedicated to that course and that course alone). If the junior biology course has 36 laboratory sections that need to be scheduled it is easier to just use one room of the four in the algorithm and schedule nine laboratory sections. If the nine scheduled laboratory sections are each placed in the four rooms that is has available to them then the original 36 sections are scheduled. This 'trick' is used in a number of places, so often it is the case that when a course appears, in the program, to have five or more sections to be scheduled, in reality there may be as many as 30 or 40 sections.

If the original test of the *Same* variable indicates that it is true then a test is made to determine whether or not sections i and j are of the same instruction type. If they are, and they are not lecture instruction types, a value of false is returned as constraints between two sections of the same course type and instruction type are not honoured thus giving the algorithm more flexibility. This flexibility is not extended towards lecture sections in this case. It is deemed important that if a course has several lecture sections, scheduling them at different times would be advantageous to the students thus giving more flexibility in their enrollment options. It is usually the case that a course has fewer lecture sections available than tutorial or laboratory sections. If the two sections are not of the same instruction type (or they are both lecture sections) then a value of true is returned indicating this constraint will be honoured. More flexibility is offered to students if the different instruction type sections contained within a course are not scheduled in overlapping timeslots. Indeed, this constraint *must* be honoured in the case of single section instruction types.

One more set of tests is carried out. If the following set of tests are all true then

a value of false is returned:

1. Section j is of a laboratory instruction type.
2. Section j has more than four sections to be scheduled.
3. Section i is of a laboratory or tutorial instruction type.

It is deemed that the fact there are many laboratory sections of course j available then the conflict between i and j can be overlooked in this case. The same test is performed again switching i for j.

Backtracking

The third version of the **Conflict()** routine is not used in the labelling of sections for backtracking purposes. A separate function has been created in order to do this.

If a section, say i, cannot be scheduled for whatever reason then backtracking is the next step in the algorithm. Section i will contain a list of all of its timeslots that are not available. Each timeslot that appears in this list due to the **UpdateConflicts()** routine will have a record of which already scheduled section caused it to be removed from the list of legal timeslots. Other timeslots that appear in the not available list are there due to the fact that they have already been tried but have been discarded in favour of another timeslot (if there were any left to try) because this section has already been the target of backtracking. The list of not available timeslots is scanned and the closest scheduled section that caused a timeslot, say t, to be removed in the **UpdateConflicts()** routine is selected as the section to which to backtrack, say j (closest value to i). Since it is section j that caused timeslot t to be removed from the list of available timeslots for section i, rescheduling section j using a different timeslot should, it is hoped, allow timeslot t to become available to section i. If no such section exists then the algorithm will backtrack to the previously scheduled section (section $i-1$). Of course, if at any time the algorithm finds it backtracks to section -1 then no solution has been found[9].

[9]This is not to say there is no solution at all. Due to the probabilistic tests made it is possible, although unlikely, that the algorithm is extremely unlucky and no solution is found even though solutions may exist.

4.2.7 Probabilistic Conflicts

Several tests in the **Conflict()** routine, as outlined in the previous section, had a probabilistic check as to whether or not the constraint between two sections would be honoured. This was used in the case of a potential conflict between a multiple section laboratory component of a course and a single section instruction type or a multiple lecture section. These were identified as somewhat troublesome sections and as such the probabilistic approach taken was used to deal with this problem. A look at the **calendar.dat** file offers a hint of what may be done to help solve this problem. Consider the piece of a **calendar.dat** file that is reproduced below:

```
N W 75
C       1st year/2nd term biochemistry
P       BIOL233   CHEM203   MATH211   PHYS203
P       BIOL233   CHEM203   MATH211   PHYS233
P       BIOL233   CHEM203   MATH253   PHYS203
P       BIOL233   CHEM203   MATH253   PHYS233
```

From this section of the file it must be determined how important the constraints contained within are. The first line in this part of the file defines it as a new *clan* for the winter term that should have about 75 students. The next line is a comment where the user can put a remark about the information contained therein. The next four lines contain possible combinations of courses that should be free for students to take. Ideally, for example, the first one implies that it should be possible for a student to take BIOL233, CHEM203, MATH211 and PHYS203 in the same term so this implies that the four courses should have no conflicts between them so that this combination is possible. In this example each line that lists a set of courses is a possible combination of courses that should be available in the winter term that a student in 1st year 2nd term biochemistry should be able to take. Corresponding to each pair of courses in each row of this list[10], there would be a '1' in the appropriate row and column of the matrix C_{cm}. In practice, for reasons previously mentioned, it is not possible to create a timetable that strictly honours every conflict in the C_{cm} matrix.

[10]Each set of four courses that are potential enrollment patterns in this case.

Some method for prioritizing the constraints in the **calendar.dat** file needs to be developed. A *probabilistic* approach was adopted in order to make an attempt at evaluating how important constraints are between various sections and probabilities were assigned to them to determine the chance that a particular constraint is honoured.

Since it has been identified that the troublesome sections are: large multiple laboratory sections (of a given course), single section courses, and lecture sections the probabilistic conflict check was introduced to try to solve this problem by honouring conflicts between *some* of these sections, but not all.

The weighting scheme for deciding the probability of a constraint being honoured between two sections is derived from the number of times the two courses appear in the **calendar.dat** file, as well as the number of times the two courses appear together in a single line in the file. In general, the more times the two courses appear together in a line the more important the constraint.

A *probability matrix* (PM) is constructed that contains the various probabilities that constraints between various courses will be taken into account. The value contained in each entry in this matrix is:

$$PM[i,j] = \frac{NumberOfTimesInCalendarTogether(i,j)}{NumberOfTimesInCalendar(i)} \tag{4.2}$$

Each entry in the matrix has a value in the range of $[0,\ldots,1]$. The constraint between courses i and j has a higher probability of being honored if the number of times they occur *together* in the same line in the **calendar.dat** file approaches the number of times course i appears in the file also. Notice that the value in the PM matrix is not symmetrical so it is often the case that $PM[i,j] \neq PM[j,i]$ since the numbers of times that i and j occur in the **calendar.dat** file are not in any way connected. If the piece of the **calendar.dat** file shown earlier is considered to be the entire file then $PM[BIOL233, CHEM203] = 1$ and $PM[BIOL233, PHYS203] = 0.5$. An example where the symmetric values are not equal would be the values of $PM[BIOL233, MATH211] = 0.5$ and $PM[MATH211, BIOL233] = 1$. So $PM[i,j]$ can be thought of as the importance of the constraint with respect to course i. In the case of MATH211 and BIOL233 in the above example it is clear that the constraint is more important to the course MATH211 as every instance of it in the **calendar.dat**

file also has BIOL233 whilst from the point of view of BIOL233 MATH211 only occurs half of the time so is deemed less important. This is reflected by having the two different probabilities in the matrix. In practice this probabilistic approach appears to work very well.

This probability matrix gives courses that occur frequently together a higher probability that constraints between them will be honoured whilst the courses that do not occur frequently together are not are as likely to have their constraints honoured thus assigning a priority as to how important some constraints are to be taken.

Some course pairings can have a low value in the probability matrix. A minimum value of 0.3 is used in the matrix PM for any two courses that have a calendar conflict.

4.2.8 Other Versions

This algorithm evolved over time and there were other approaches used that did not bear good results. For example, the first approach to solving the MTT problem concentrated more on the "tightness" on the availability of the rooms to the courses. The courses with a large number of laboratory sections were quickly identified as problematic with this approach as there was little or no flexibility in where the laboratory sections could be placed. This emphasis on the 'tightness' has changed to the prioritizing of conflicts between courses to take into account single section courses and the large laboratory sections that can be problematic.

4.3 Implementation Details

The heuristic algorithm was initially programmed in a UNIX environment using the C programming language. It was written for portability and has since been moved to a PC platform with little difficulty[11]. The testing was carried out on a Pentium Pro PC using Windows-NT.

A typical running time for the heuristic algorithm varied from a few seconds to several minutes. Due to its probabilistic nature, it is possible the algorithm will

[11]This is due to the Author's relocation to the United Kingdom when final testing was being carried out. The Author was allocated a PC in the department so translating both the Heuristic algorithm and the Genetic Algorithm to the PC was necessary and convenient. This also serves to demonstrate the importance of writing portable code!

make some poor decisions and get caught in the resulting combinatorial explosion. Fortunately, the backtracking method outlined earlier has proved capable in coping with any poor choices made.

4.4 Summary

The heuristic method described here is quite complex in parts and tries to take into account the intrinsic difficulties that arise with various courses in the scheduling process. The algorithm developed tries to treat the conflicts between different courses within the context of how *crucial* the conflict is between them. This, of course, depends on the courses in question, how many sections they have as well as these sections' instruction types.

The actual runtime of the algorithm on the biology/chemistry set of data varies from a few seconds to a few minutes due to the probabilistic element in it. In general the results generated with this approach are superior to the actual schedule used.

Chapter 5

Application of Genetic Algorithm Method

The second approach used to attempt to solve the master timetabling problem is a genetic algorithm approach. As was seen in Chapter 3 genetic algorithms try to mimic the process of evolution in nature. [CRF94a] observes that there are three core steps in using a GA for a timetabling problem.

1. Decide how to represent a timetable as a chromosome.
2. Decide how to measure the 'fitness' of a timetable.
3. Decide on appropriate recombination (crossover) and mutation operators.

There are two approaches to the first step involving the representation of the chromosome. The first is a *direct* representation and the second is an *implicit* representation [CRF94a]. This will be discussed in more detail later. The second step involves deciding on a fitness function to measure the quality of a particular solution (chromosome). The value of this function should represent how desirable a solution is. The third step involves the definition of what genetic operators are to be employed such as crossover or mutation.

These steps will be described in more detail as to how they will be applied to the MTTP in the following subsection.

5.1 GA Representation of the MTTP

5.1.1 Chromosome Representation

For scheduling problems there are two approaches to chromosome representation that seem to be present in the OR literature. These are direct representation and implicit representation. Direct representation is the approach used for the MTTP in this thesis. In this problem there are only two basic properties for any particular course section that need to be considered. They are:

1. The time the course is held.

2. The room in which the course is to be held.

The chromosome representation is fairly straightforward and is similar to representations used by others to solve similar scheduling problems [RCF94b, Fan92]. The representation consists of each *gene* representing a course section where a gene is composed of two elements (integers) that represent the timeslot in which a course section is to be scheduled, as well as the room it will be placed. Work done by others tackling this problem sometimes also takes into account the lecturer of each course. In the timetables considered within this thesis many of the courses are not assigned a lecturer ahead of time (and even if they are, the instructor assignment can change) therefore this was not seen as an important factor with this particular research; it could be added into either method presented in this thesis if it was needed.

BIOL231 L01		...	ZOOL 273 B15	
MWF 1200 50	ST 150	...	T 1100 170	SA 122

Figure 5.1: Chromosome Representation

Figure 5.1 shows what this chromosome representation would look like. In this example the first gene represents BIOL231 L01 which is assigned to take place on Monday, Wednesday and Friday at noon for 50 minutes. All of the other courses to be scheduled are contained in this chromosome up to the last one which is ZOOL 273 B15 in this example. Many of these chromosomes, each representing a potential

solution, form the *population* of the GA from which the best solution is hoped to evolve.

The terminology to be used with the chromosomes is the same that was used with the heuristic method. Each gene is referred to as a *section*. Each section belongs to a particular *course* (for example BIOL 231) and *instruction type* (Lecture, Laboratory or Tutorial). A course of a particular instruction type may have several sections. For example BIOL 231 may have 3 lecture sections to be scheduled in the timetable.

Clearly any optimal solution for the MTTP can be represented with this chromosome representation. An optimal solution, O, can be mapped to a chromosome simply by copying the rooms and timeslots over to their corresponding genes. With the crossover operators being used it is also possible for such a solution to be generated. The crossover operators examined earlier (one-point, two-point and uniform) can clearly introduce the good characteristics of one chromosome into another. For example, if a good subschedule for an adjacent set of sections in a chromosome exists then a genetic operator, such as two-point crossover, can essentially *cut out* this good portion of a chromosome and insert it into another chromosome (if the two crossover points fell on either end of the good portion contained in the original chromosome). Similar arguments could be made for one-point and uniform crossover. Crossover can bring desired traits from one parent to another in the population in order to bring solutions closer to an optimal one. Mutation can introduce missing pieces of genetic material that would enable a solution to become closer to the optimal one. Unfortunately, it is probably the case that the set of optimal solutions with respect to the entire solution space is probably quite small. This is a problem that most GA's face.

An implicit representation of this problem would be somewhat more abstract than the direct representation outlined above. Implicit representations are exemplified by a chromosome that is a permutation of events (or in the MTTP, sections). Evaluation involves the use of a heuristic, say H. This heuristic would be applied to each event in the order that it appears in the chromosome. As an example, consider the chromosome "abc" where a, b, and c are sections to be scheduled. The heuristic would schedule section a, then section b and finally section c given the permutation of the sections in the chromosome. A one pass heuristic would be the obvious approach for chromosome evaluation. This approach is mentioned in [CRF94a] but does not seem to be actually used in most GA's described.

5.1.2 Fitness Function

The fitness function is an important consideration in the design of a GA as it assigns a value representing the quality of a solution. It is these fitness values that are used in the process of probabilistically selecting which chromosomes should participate in the creation of offspring. The fitness function is one of the forces that pushes the GA in the direction of better solutions if it is chosen carefully. The fitness function is also the most problem specific part of the traditional GA formulation as it must incorporate problem specific knowledge in order to be successful in identifying both superior and inferior solutions.

A popular fitness function used in the GA literature is to calculate the fitness inversely proportional to the number of constraints violated in the schedule. Each violated constraint will have a weight attached to it that represents how 'undesirable' that feature is in the timetable [BBM93a, CRF94b, Ric95]. These weights can be thought of as *penalties*.

Let C represent the set of constraints, and let N_C be the cardinality of C. Let w_j represent the weight carried by constraint c_j when violated. Also let $v(c, x)$ be the number of times constraint c is violated in chromosome x. Then the fitness function can be defined as:

$$f(x) = \frac{1.0}{1.0 + \sum_{j=1}^{N_C} w_j v(c_j, x)}, \quad \text{where } c_j \in C. \tag{5.1}$$

The function $v(c, x)$ would involve comparing each pair of sections in the chromosome in order to count the number of times that constraint c is violated in chromosome x. Because of this the fitness function it is an $O(n^2)$ calculation[1] where n is the size of the chromosome (number of sections). This is a somewhat expensive but unavoidable operation in the GA. Most of the GA's runtime is spent calculating the fitness of the chromosomes in the population.

From the lessons learned in the construction of the heuristic presented in Chapter 4 the constraints are categorized depending on how critical they are to producing a satisfactory timetable.

[1] There are roughly n^2 steps needed in $v(c, x)$ to compare each pair of sections in chromosome x for potential constraint violations. That is, the first section would need $n-1$ comparisons, the second section would need $n-2, \ldots, 1$. The total number of comparisons needed would be $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$.

The constraints taken into account by the fitness function (elements of C) are as follows:

1. Two sections cannot be placed in the same room in the same term in overlapping times. (This is $c_1 \in C$).

2. Timeslot conflicts should be avoided between certain conflicting sections[2]. These fall into several categories depending on the course and instruction types of the sections involved. They are:

   (a) Conflicts are not allowed between two *lecture instruction type sections* with one of them being a single section lecture type (c_2).

   (b) Conflicts between a *lecture instruction type* section (l) and a *tutorial or laboratory instruction type* section (t). There are four constraints restricting this type of conflict.

      i. Conflicts should be avoided if both sections are of the same course type and t has more than four sections and l has more than one section (c_3)

      ii. Conflicts are not allowed if both sections are of the same course type and l is a single section (c_4).

      iii. Conflicts are not allowed if both sections are single section types (c_5).

      iv. Conflicts should be avoided between l and t if t has fewer than seven sections and none of constraints c_3, c_4, and c_5 have been violated by the two sections in question (c_6).

   (c) Conflicts between two *laboratory* or *tutorial* sections. There are only two cases considered:

      i. Conflicts are not allowed if both sections are of the same course type and either of them has only one section (c_7).

      ii. Conflicts should be avoided if both sections have fewer than five sections and conflict c_7 is not violated by the two sections in question (c_8).

[2]Conflicting sections in the sense that they have a value of one in the calendar conflict matrix.

There is one more constraint that is not mentioned above. It is an additional constraint that comes into effect when there are two conflicting sections that are of particular importance.

Recall that in the heuristic algorithm in the previous chapter a matrix PM was constructed containing entries with values between zero and one. This matrix, the probability matrix, was used to decide how crucial a constraint was between two sections. This identical matrix is used in the GA for similar reasons. If two sections are being compared for conflicts, say i and j, then the values of $PM_{(i,j)}$ and $PM_{(j,i)}$ are recorded[3]. If either of them is $\geq .9$ then a conflict is said to be *crucial*. The violation of constraint c_9 can be defined as an instance of the violation of any one of constraints c_2, c_6, or c_8 that are between sections that also have this *crucial* conflict property. For example, if there is a violation of constraint c_2 in a chromosome *and* the two conflicting sections in question have the crucial conflict property outlined above then this is also a violation of constraint c_9. Essentially the two lecture sections in question (in this case) have violated two constraints as far as the GA fitness function is concerned. The violation of the c_9 constraint can be seen as an additional penalty on the violation of constraints c_2, c_6 and c_8.

The types of constraints outlined above fall into several categories. Some of them are to be treated more seriously than others. Each constraint type above (c_1 through c_9) has a different *weight*, w_j, associated with it as found in the fitness function in equation 5.1. Weights should be chosen so as to correspond to the seriousness of the various conflicts[4]. The weights used for the violation of the previously defined constraints can be found in Table 5.1.

Conflict	Weight	Conflict	Weight
c_1	42	c_6	12
c_2	42	c_7	42
c_3	12	c_8	10
c_4	42	c_9	7
c_5	42		

Table 5.1: Conflict Weightings

[3]Recall, the matrix PM is not symmetric.

[4]A violated constraint can also be referred to as a conflict.

At this point it should be stressed that *other* constraints were experimented with but the ones above were found to give schedules that were useful, or were easily modified to be useful, consistently. The other constraints were for the most part dropped. In the earlier versions of the GA more constraints were used in the fitness function. Constraints for *any* conflicting sections were initially implemented. Some were weighted as more serious than others; for example a conflict between laboratory instruction type sections for courses that have more than six laboratory sections to schedule were weighted with a penalty of one. It was these types of conflicts that were very common (and had a low penalty in the fitness function) that seemed to cause a problem. In general the weights for these constraint violations were quite low (values of between one and five). Due to the fact that the violation of these constraints was so common the cumulative effect of many low weighted constraints interfered with the more serious constraints (such as room conflicts). If the serious conflicts' weightings were increased to combat this effect the solution would tend to be pushed into a poor local minimum early in the GA process. Unfortunately it was difficult for the GA to come to a good solution with all these constraints being taken into account. Different weight values were tried but it was found that the algorithm worked well if the constraints c_1 through c_9 were present with the values shown in Table 5.1. The heavy weight values were found to be large enough to drive the GA to a good solution but not so heavy as to force it into a local optimum too early.

As mentioned earlier, the weights used for these constraints reflect how important they are with respect to a good solution. The constraints with the weight of 42 are seen to be very important to a good solution whilst the weightings of 12 and 10 are seen as important but not as critical as the conflicts having a weight of 42 (constraints c_1, c_2, c_4, c_5, c_7). The weight of seven for the c_9 conflict is used as an additional penalty if the sections conflicting are seen to be *crucial* as previously described. Whilst the above set of constraints chosen is not perfect they seem to be the constraints with the largest influence on the general quality of a solution.

5.1.3 Genetic Operators

Two genetic operators are used in this application to search the solution space. They are *crossover* and *mutation*.

Crossover Operators

Three types of crossover are used in the genetic algorithm. The classic one and two point operators are used as well as uniform crossover [Sys89, Mic92, Ree93]. These were outlined earlier in Section 3.3.3. The only difference is rather than swapping bits between the parent chromosomes, genes are swapped (timeslots and rooms).

Mutation Operator

The mutation method used is to assign a low probability that one (or both) of the values found in a gene will be changed. The room or timeslot value is changed to a random value taken from the timeslot or room category of the section in question.

5.2 Algorithm

The algorithm used in this genetic algorithm is a fairly straightforward one that can be found in most of the GA literature. A few extra features were found to be needed which also appear in the algorithm. The algorithm is outlined in pseudocode in Figure 5.2.

```
procedure GeneticAlgorithm() {
   CreateInitialPopulation();
   DoPreSchedule();
   EvaluatePopulation();
   for (i = 0 ; i < Iterations ; i++) {
      Reproduce();
      UpdateCurrentPop();
      Mutate();
      EvaluatePopulation();
   } /* FOR */
   PrintSchedule();
} /* GeneticAlgorithm */
```

Figure 5.2: Genetic Algorithm

As can be seen in the pseudocode the algorithm begins with creating an initial population of chromosomes (solutions). Solutions are created randomly with timeslots

and rooms selected at random from the available room and timeslot categories. This gives a poor initial result but, on the other hand, gives a good diversity of genetic material from which the process can be started.

After the population creation stage the option to *preschedule* selected sections is available. Sections that are prescheduled do not have their timeslots or rooms modified at any point in the algorithm and are in place such that other sections can have the constraints in the GA checked against them. Generally it is courses that have large numbers of laboratory sections to be scheduled that are prescheduled. There is often little choice in where to place these laboratory sections so prescheduling them makes sense and saves the GA from having to place them.

It should be stated that the GA was found, surprisingly, to be slightly more effective if the lecture sections corresponding to the prescheduled laboratory sections were prescheduled also. Since the laboratory sections have little flexibility in where they can be placed the same can be said, to a lesser degree, for the lecture sections (with respect to their laboratory sections). Most of these courses only have one or two lecture sections to schedule so they are prescheduled to not overlap with any of their laboratory sections (although if there were several lecture sections to be placed it may not be a poor choice if they did overlap).

Next an initial evaluation of the population is to be made. This is for the reproductive step as the fitness of each chromosome must be known.

The main loop in the GA is entered at this point and reproduction is the first step. A probability is assigned as to how likely crossover is to take place for any particular pair of parents. This is known as the *crossover probability*. The roulette wheel parent selection method is used as outlined in Figure 3.6. Once two parents have been selected a random value is generated in the range $[0, 1]$ and checked against the crossover probability. If crossover is to occur the parents are mated with one of the three crossover methods that are available[5]. If crossover does not take place the two selected chromosomes are copied into the new population; otherwise the newly created chromosomes (via crossover) are. At the end of the reproduction routine the *best* solution found so far is copied into the new population[6] and a random solution is inserted also (to introduce some potentially new genetic material). The entire popula-

[5]Single point, two point or uniform crossover. This is a parameter for the program.
[6]Sometimes referred to as an *elitist* model.

tion is replaced with this process. It is at this stage in the GA that better solutions are generated or reasonable solutions are carried over from the current generation into the new one. With the roulette wheel parent selection process, on average, the stronger members of the population have their genetic makeup mixed with one another to, it is hoped, produce better offspring (solutions).

After reproduction and the updating of the population has been carried out the mutation function is called. This operator examines each gene in each chromosome and with a small probability, referred to as the *mutation probability*, it may randomly change the value of the room or timeslot (or both) in the gene being examined. The values that a gene can take will be limited to what is legally available to it in that gene's timeslot and room categories[7] (conflicts nonwithstanding). The purpose of this step is to try to maintain some genetic diversity in the population. This helps to ensure that more of the solution space is examined and helps to try to avoid local minima.

These steps in the main loop are repeated however many iterations the user desires although after two or three thousand iterations the population has usually converged such that all the solutions are fairly similar and improvement is unlikely.

5.2.1 Parameters

As has been alluded to above, there are several parameters that must be decided upon in the GA process.

Crossover Operator

When the GA is invoked a selection as to which crossover operator to be used is passed as a parameter to the program.

Crossover Probability

Many crossover probability values were tried. Tests were carried out using a constant crossover probability. Values from 0.10 to 0.90 in 0.10 intervals were tried.

[7]Since each gene is a section.

Mutation Probability

In this GA implementation the mutation changed over the duration of the GA. Common values for mutation in the literature tend to be very low, usually around .001 to .03. The mutation probability used started at a value of 0.002 and finished with a value of 0.05. The mutation probability was increased by a constant amount after each iteration of the GA process; this constant value was calculated from the initial and final value of the mutation probability along with the number of iterations to be used in the GA process. Mutation probabilities should be low as the function of mutation is intended to bring some new genetic material into the solutions space. A high mutation probability would probably not fill this role and would have a good chance of disrupting good solutions already present in the population.

5.3 Implementation Details

Like the heuristic algorithm in the previous chapter the Genetic Algorithm was initially programmed in a UNIX environment but C++ was used instead of C (for a cleaner implementation of chromosomes and populations as the C++ class structure makes this convenient.).

For reasons mentioned in the previous chapter it was found convenient to port the program over to a PC environment and this was done with little difficulty.

The runtime of the algorithm is a direct function of the number of generations it is expected to do as well as, obviously, the problem size. The fitness function is the most expensive operation in the GA and has a complexity of $O(n^2)$ where n is the chromosome size (number of sections to be scheduled). For 2,500 iterations (the value used for testing purposes) the program was found to take about one and a half hours to run on a Pentium Pro 300 PC running under Windows-NT. This is very slow compared to the Heuristic Approach[8].

[8]It may also be useful to point out that evolution in nature, upon which GA's are based, can take millions of years. GA's are a form of 'solution evolution' so perhaps it is not surprising it takes as long as it does!

5.4 Experiments

With the fitness function decided upon the next step was running some experiments using various parameter values.

Mutation was probably the hardest to decide upon as it is difficult to measure how well it performs because it is a subtle operator. Examining the role it has in the GA process helped to arrive at a decision on its value. In [Ree93] (in which Reeves cites [Dav91]) it is observed that crossover is important at the beginning of the GA process when the population is diverse. As the solutions (chromosomes) begin to converge it is important to increase the chance of finding different solutions (introduce some genetic diversity). This is where mutation is most useful.

Considering this fact it seems sensible to have a changing mutation rate. As stated earlier the value chosen was to start from 0.002 and increase, in a linear fashion, to the final value of 0.05. With these values mutation starts off slow but introduces more genetic material late in the GA process when it is most needed.

Both the crossover rate and crossover types required more direct experimentation in order to decide which ones work the best. Several values of crossover rate were tried. Values from 0.10 to 0.90 were used. To illustrate which crossover rate and crossover type is better, averages were taken of the fitness from 10 trial runs on the Biology/Chemistry data using the above crossover probabilities with all three types of crossover. This can be found in Table 5.2. Figure 5.3 shows the results from the table graphically.

Crossover Rate	2-Point	1-Point	Uniform
0.10	21.93	21.97	24.17
0.20	25.60	23.94	21.47
0.30	21.31	20.66	19.01
0.40	21.91	21.80	19.65
0.50	20.06	20.04	19.01
0.60	18.91	17.50	17.33
0.70	18.46	18.54	14.59
0.80	17.53	15.33	12.83
0.90	16.68	13.52	11.14

Table 5.2: Average Fitness with Three Crossover Operators

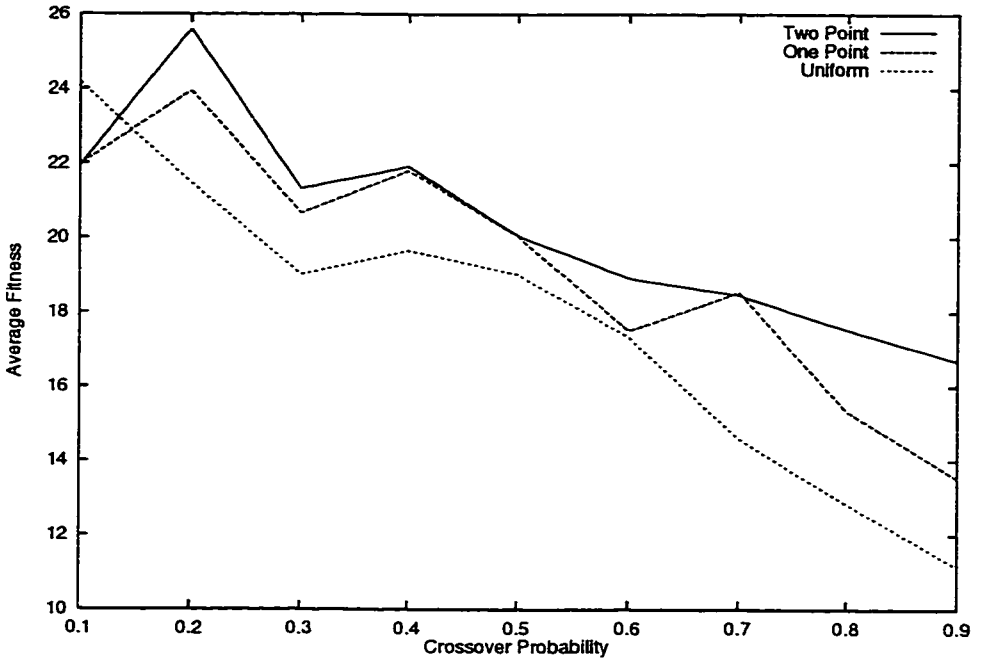


Figure 5.3: Average Fitness vs Crossover Probabilities/Types.

From the information in Table 5.2 and Figure 5.3 it is shown that 2 point crossover used with a crossover rate of .20 seems to be the best set of parameters with regards to fitness. In fact, both the average fitness values for 2-point and 1-point crossover peak at a crossover rate of 0.20 (and they peak again, although at a lower average fitness value, at 0.40). This is somewhat surprising as crossover rates generally tend to be quite high in the literature. For example [Fan92] uses an initial crossover rate of 0.70. Little mention is made in the the scheduling GA literature examined as to what testing was done to arrive at these values. It is also interesting to note that uniform crossover seems to be a popular approach for scheduling using GA's as can be seen in [Ric95, RCF94b] whilst in the experiments carried out in this thesis clearly show uniform crossover to be the worst choice overall (except, possibly, with the exceptionally low crossover probability of 0.10). Whilst the crossover probability used in this work seems low it is worthwhile to note that low values have been used elsewhere. In many of the examples of GA formulation given in [Mic92] low crossover probabilities are proposed (values of 0.05 to 0.40 in some cases). Unfortunately, the reasons for this are not always clear. To some degree it will come down to experimentation with the

problem at hand. It should be noted, however, that the problem being tackled here is much more highly constrained than many of the problems seen elsewhere. However, to be fair, most of the other GA methods examined were course based rather than student based scheduling problems or exam scheduling problems. [Ree93] states that *robustness* is one of the advantages to using a GA. He goes on to state that GA's can often produce acceptable results over a wide range of parameter settings which is the case with this work also. Although 0.20 seems to be the best crossover probability and two point crossover seems to be the most useful of the crossover operators it is the case that solutions generated with different crossover probabilities and different crossover operators were also acceptable some of the time (although not as frequently).

Due to the results from Table 5.2 a crossover probability of 0.20 as well as two point crossover and the linearly increasing mutation rate starting at 0.002 to 0.05 were used to generate the results reported in Chapter 6.

5.5 Summary

The genetic algorithm used is fairly straightforward and simple in that it, in many ways, follows the outline of a classic GA as found in the literature. The results it generated were found to be good but not as good as the heuristic method. Both methods tried did generate schedules of a better quality to the actual ones used as will be seen in the next chapter.

The next chapter examines the schedules produced with both methods presented in this thesis.

Chapter 6

Results

Although the two methods used to approach this problem are quite different, the results from both were quite good. The results were by no means perfect; however, both methods could generate schedules better than the actual schedule. The heuristic was the more reliable of the two methods as it created better schedules than both the actual one and the GA produced ones.

Before the results can be examined, the method by which they are evaluated will first be discussed.

6.1 Schedule Evaluation

The program used for the evaluation of the schedules produced by the algorithms presented in this thesis was created by Joshua Chen who worked under Dr. Colijn. The algorithm itself was taken from [Col73]. This program generates fictitious students based on calendar information as well as historical information. This information is provided in the **calendar.dat** file as well as the actual student enrollment data from the 1993-1994 academic year. The **calendar.dat** file is explained in more detail in Appendix A.1. This set of fictitious students is then sectioned (i.e. enrolled in the requested courses, and, if at all possible, assigned to conflict-free sections of these courses) using the schedule being tested. The details of it won't be presented here[1]. A brief description of the sectioning method will be presented next.

[1] The sectioning program used was created from a CPSC 502 project.

6.2 Student Sectioning Method

The method for generating and sectioning fictitious students does not take into account various factors that may influence real students when making course (or section) selections. For example, the fictitious students are as likely to be assigned to an 8:00AM section as to a 1:00PM one. The sectioning program generates these fictitious students with course requests. Next the sectioning program assigns these fictitious students to sections. Real students choose not only the courses, but also the sections. The sectioning method merely tries to test the feasibility of a schedule through this process. It identifies any sections that have a serious conflict as well as any sections that have an unusually low enrollment which may imply a more subtle conflict that is present in the schedule.

The first step in the sectioning process is to generate a list of fictitious students. Each programme listed in the **calendar.dat** file has one or more possible enrollment patterns (recommended set of courses to take). The actual enrollment data from the 1993-1994 academic year is scanned and the number of students that were enrolled in a recommended set of courses is counted. Each programme in the **calendar.dat** file also has a suggested enrollment figure supplied with it. This will be the number of students in the particular programme being examined that will be created. The number of students for each recommended set of courses in a programme is determined by the proportion of the students in the historical data that were enrolled in them. An example will help illustrate. Take the following sample from the **calendar.dat** file.

```
N F 5
C 2nd year/1st term honours biochemistry
P       BIOL311   BOTA225   CHEM354   CHEM410
P       BIOL311   ZOOL273   CHEM354   CHEM410
N W 5
C       2nd year/2nd term honours biochem
P       BCEM441   CMMB301   CHEM354   CHEM410
```

The above section of the **calendar.dat** file is for second year honours biochemistry students. The enrollment was expected to be approximately five students. The 1993-1994 student enrollment file is examined and five fictitious students are generated who

are enrolled in this programme. This is done using the proportion of the students taking the first programme pattern (involving BOTA225) and the second programme pattern (with ZOOL273). For example it may be the case that only one will be enrolled in the first fall programme for second year first term biochemistry and four will be enrolled in the second if the proportion of students historically enrolled in these programmes is approximately 20% and 80% respectively. For the winter term all five students will be enrolled in the four courses listed under 2nd year/2nd term honours biochemistry in the **calendar.dat** file. It should be noted that there is only one enrollment pattern for this particular term/programme. It is in this fashion that the fall and winter course selection for fictitious students is carried out using the information kept in the **calendar.dat** file as well as the 1993-1994 student enrollment file.

Essentially the sectioning algorithm processes the fictitious students one at a time. Each student generated will have a list of courses that are desired. These lists are derived from the different rows in the **calendar.dat** file as outlined above. These entries are derived from the calendar requirements for the different years in each department (Biology, Chemistry, etc). First the sectioning algorithm separates out the instruction types for the student being sectioned (so that for courses with lectures, labs and tutorials there will be three entries instead of one). After sorting these entries from the fewest sections to the most sections the algorithm then goes through these sections one by one, assigning in each case the conflict-free section of lowest enrollment that is available at that time. If, for a particular entry, there are no conflict-free sections available, the algorithm backtracks (in a Brelaz-like fashion) to an earlier entry for the same student and tries a different assignment for it at which point it goes forwards again. If the backtracking process takes the algorithm all the way back to the first entry for the current student then no conflict free assignment is possible for this student. That is to say that this particular student cannot be sectioned. The full details of this process are available in [Col73].

6.2.1 Information Given by Student Sectioning

The information given by the sectioning process consists of identifying which courses scheduled have serious conflicts. A list of courses that have a low (or even zero)

enrollment is also provided as this indicates some, possibly more subtle, conflicts at work.

The most important conflicts identified by the sectioning program are what are known as *direct* conflicts. Generally these are conflicts between different lecture sections that should be conflict free and are serious enough such that it is impossible to schedule students to them as required in the **calendar.dat** file. That is to say that they have a constraint listed in the **calendar.dat** file that has not been honoured and, in this case, should have been. Lectures and conflicting lab sections may appear in this category as well if they are tightly constrained. For example if in the **calendar.dat** file a course always appears in the same row as another course then these two courses can be thought of as having a *tight* conflict. It is then clearly important that they are assigned non-conflicting times, especially if they have a single section instruction type. Regardless, the courses that have a direct conflict generally appear in the same line at some point in the **calendar.dat** file. Of particular importance are single section lecture instruction type courses as conflicts with them are unacceptable unless the conflicting courses are of a multiple section instruction type (for example a laboratory instruction type section that is a part of a course with many laboratory sections). Even in this case conflicts between them should, at least, be partially honoured.

The second piece of information given by the sectioning process is a list of courses with an unusually low enrollment. The sectioning algorithm tries to enroll students into sections in the schedule such that they are evenly spread out but it seems inevitable that some clustering does appear and a side effect of this is that some sections tend to have a lower than expected enrollment (or even a zero enrollment). If some sections of a course have much lower than expected enrollments, this can be an indication of a (possibly subtle) problem in the timetable.

The sectioning program produces a log file of what it has done. This includes a list of the fictitious students generated along with the courses they are to be enrolled in. Other information supplied includes a list of directly conflicted sections, the sections they conflict with, a list of sections with an empty or low enrollment as well as sections that conflict with them.

6.3 Results of Sectioning Program on the 94/95 Schedule

The main objective of this research was to be able to produce solutions generated with the heuristic and GA that are of a quality similar to, or better, than the actual schedules used in practice. The schedule that is to be examined is the actual schedule used for the Biology/Chemistry master timetable problem for the academic year 1994-1995. It is this schedule that will be used as a basis for comparison for the schedules generated with the heuristic method and the GA method.

One surprising result at this point is that the actual timetable from the 1994-1995 academic year has one direct conflict. The sections CHEM354 L1 (a full year course) and ZOOL273 L1 in the fall semester have conflicting times. They both are held on MWF at 8:00AM for 50 minutes. CHEM354 and ZOOL273 also only have one lecture section each. In the University of Calgary calendar on page 304 it has CHEM354 and ZOOL273 listed as a recommended enrollment for second year biochemistry [Cal94]. The reason this conflict exists is unknown but CHEM354 is listed as an alternative[2] for CHEM350 so this year it must have been known that CHEM350 was the only choice contrary to what the calendar implies.

Generally, the number of direct conflicts as well as the number of empty or low enrollment sections will be used as a metric to measure the quality of solutions.

At this point it is interesting to observe that, superficially, many timetabling problems have similarities between them. Unfortunately it is difficult to compare the quality of two timetables between different institutions because of many practical details that are almost guaranteed to be different from institution to instition and, often, from department to department. An example of this would be the fact that at the University of Calgary Biology courses are not allowed to have some of their lab sections take place on Mondays while other departments have no such constraint. It is details such as these that can have a substantial impact on the effectiveness of the methods employed in this thesis.

Another, perhaps initially surprising, result is the number of empty or low enrollment sections produced by the sectioning program when applied to this schedule.

[2]CHEM350 is recommended for majors but CHEM354 is also an acceptable choice.

Bear in mind that the sectioning program is using the actual timetable but with fictitious students. The list of these empty/low enrollment sections found in the 1994-1995 academic year schedule by the sectioning program can be found in Table 6.1. There are several reasons for these empty/low enrollment sections which will be examined next.

Some explanation for the empty/low enrollment sections can be found in the fact that while all fictitious (generated) students follow the enrollment patterns that various departments advise, not all real students do. Indeed, there are many cases, in practice, that could have some influence on this. For example some students may have special permission to enroll in a somewhat unusual combination of courses or perhaps a student only passed some of the prerequisites needed and therefore has an unusual timetable selection for the following year. The **calendar.dat** file contains some estimates of enrollments expected in various programmes that may also be inaccurate in practice so the sectioning program might be unintentionally biased towards enrolling more students in courses than what might otherwise be expected or actually achieved.

Another reason for the empty/low enrollment sections is that some of the sections are co-requisites for programmes in departments not considered here. For example it is the case that degrees such as physics or geology may require the students to take some chemistry courses. The empty sections found here will be the ones most likely used by non Biology/Chemistry majors or students with unusual enrollments.

To understand why some of these sections have such a low enrollment consider the CHEM 350 B1 section that is listed in Table 6.1. The sectioning program was unable to schedule *any* students into this section. The sectioning program also gives some additional information which consists of sections which conflict with the CHEM 350 B1 section timewise. The sections that conflict with this chemistry laboratory section as well as the times they are scheduled can be found in Table 6.2.

Given that the CHEM 350 B1 section is scheduled to be held on Mondays at 14:00 for 170 minutes it can easily be verified that this section conflicts with all the ones in Table 6.2. Recall that two sections can only conflict when they appear together in the **calendar.dat** file. This implies that in practice it should be possible for a student to enroll in both courses. If the **calendar.dat** file is examined closer it is the case that at least one course listed in Table 6.2 appears in every line in the **calendar.dat** file

Course	Term/Inst Type/Secs	Enrollment(s)
BCEM441	W/B/6	0
BCEM443	F/B/4,8,12,13	7,7,7,6
BOTA225	F/B/2,5,10	6,8,8
CHEM330	B/B/1	2
CHEM341	F/B/5,6	0,0
CHEM350	B/B/1,7,8,9,15,16,17	0,0,0,0,0,0,0
CHEM350	B/T/1,3	10,10
ECOL313	F/B/9,11	1,1
ECOL317	W/B/6	2
ZOOL373	F/B/2,5	14,15
ZOOL375	F/B/2,5	0,0
ZOOL377	W/B/2,5	0,0

Table 6.1: Empty/Low Enrollment Sections for Actual BIOL/CHEM Schedule

Course/Term/Inst Type/Section	Time Scheduled
BIOL311/F/L/1	MWF 1600
BOTA323/F/L/1	MWF 1500
CMMB301/W/L/1	MWF 1500
ECOL317/W/L/1	MWF 1400
ZOOL373/F/L/1	MWF 1500

Table 6.2: Sections conflicting with CHEM 350 B1

that also contains CHEM350. Since the sectioning program only produces students that are taking courses as displayed in the **calendar.dat** file it is easy to see why the sectioning program does not enroll any students in it.

There are sections with an exceptionally *low*, but *nonzero*, enrollment. This is due to a similar, but less constrained, version of the empty section problem. For example BCEM443 B4 from Table 6.1 has an enrollment of 7. Similar to the CHEM350 B1 section outlined above, the BCEM443 B4 section has a list of sections that conflict with it timewise. The **calendar.dat** file contains 14 different enrollment patterns involving BCEM443. Nine of these contain courses whose lecture sections conflict with BCEM443 B4. This means there are only five enrollment patterns that can be legally scheduled using BCEM443 B4. This is why the enrollment is low but not zero.

It can accommodate some of the students generated by the sectioning algorithm but not all of them (in this case the students taking one of the nine conflicting course patterns).

Some additional information can be gathered from an examination of the historical data. Similar results to what is found in Table 6.1 are found when the actual timetable for the 1993-1994 academic year is put into the sectioning program. This in itself is not surprising as the calendar requirements for the Biology/Chemistry departments for the academic years 93/94 and 94/95 are very similar. This is also why the results are focused on the 94/95 data[3]. If the course lists of the students, from the actual 93/94 academic year, that are enrolled in CHEM350 are examined some interesting results can be found. There were 406 students enrolled in CHEM350 in total. Five of them had CHEM350 as their only course in the fall term. A more interesting fact found was that 93 students who were enrolled in CHEM350 in the fall term were not taking any of the other courses offered by the Biology/Chemistry department[4]. Many of them were enrolled in Kinesiology, Physics, Geology or even Psychology courses in addition to CHEM350. In the winter term for the 93/94 academic year it was found that 58 students who were enrolled in CHEM350 were not enrolled in any other courses offered by the Biology/Chemistry department. Many of the students were also enrolled in what could be considered optional courses such as ASTR205 or PSYC205. This clearly demonstrates that even if the sectioning shows some courses as being empty it may not be too large a concern due to the inter-departmental enrollment trends that would be present in a real schedule. Also, in practice it seems that students do not always follow the recommended enrollment patterns as found in the calendar. In examining the historical data it was not uncommon to see students enrolled in only a portion of the recommended courses for a programme whilst the students generated with the sectioning program are enrolled in an entire programme recommended course pattern. More importantly these reasons help explain why there are so many courses listed as having a low enrollment.

From this it can be seen that these empty/low enrollment sections are not a press-

[3]Because, as mentioned, the scheduling requirements for the Biology/Chemistry data is virtually identical from year to year. The only major difference between 93/94 and 94/95 is the introduction of the BIOL231 and BIOL233 courses and the elimination of BIOL201.

[4]For example BIOL/CHEM/CMMB/ZOOL/ECOL/BOTA etc.

ing cause for concern. It would be ideal to try to minimize the number of such sections in order to maximize the choice majors in these fields can have. The empty/low enrollment sections identified by the sectioning program for the actual 94/95 timetable are generally courses that have a reasonably large number of laboratory sections so it is not surprising that they are conflicting with various courses. Many students are not enrolled in the courses that are expected as they can be enrolled in different majors, part time, or even stuck "in between" years.

6.3.1 Data Used

The data that is used for testing consists of most of the courses offered by the Biology and Chemistry departments[5] for the 1994-1995 academic year. A partial list of the **calendar.dat** file as well as an explanation of its format can be found in Appendix A.1. A partial listing of the **combo.dat** file can also be found in Appendix A.2. The timeslots categories and room categories used were generated from the types of timeslots and rooms used in the actual 93/94 and 94/95 schedules. Some extra timeslots were added because the timeslots actually used in 93/94 and 94/95 do not necessarily represent the only possible ones. Some intelligent choices had to be made from experience however. For example, a few highly constrained chemistry courses[6] had their lecture section timeslot categories restricted to early morning, midday, and late afternoon. This was to avoid conflicting with their laboratory section timeslots as they tended to be placed in the mid-morning and mid-afternoon. This made sense for a few courses but generally the timeslots created for sections were very flexible and ran throughout the day. If many timeslots were restricted in this way then the user would be forcing a certain "shape" to the schedule that may prevent good solutions from being formed. This highlights that some thought has to be put into the selection of timeslot categories for courses. In total, the Biology/Chemistry data set required 341 individual sections to be scheduled (this does not include the sections taken into account with the multiple section laboratory scheduling 'trick' described earlier).

[5]This includes BCEM, BIOL, CHEM, CMMB, ECOL, ZOOL

[6]CHEM201, CHEM203, CHEM350, CHEM354, CHEM374 lecture sections were the only sections to have their timeslot categories restricted in such a manner. These courses appear very often in the **calendar.dat** file and are highly constrained.

The set of courses to be scheduled was found to be a tightly constrained scheduling problem as outlined in earlier sections. The additional constraints of scheduling courses for departments outside of Biology and Chemistry were not considered as that would make the problem very large. Options within each of the degree programs were also not considered either as it is not clear what sort of scheduling strategy should be used for these sections. Options and the pattern of options available from term to term and year to year can tend to change. As an aside, there are different kinds of options. There are recommended options, options chosen from a more-or-less restrictive list[7], 'free' options, etc. The only guide that is available at the moment would be historical data. This is prejudiced by the fact the students in the historical data were influenced by the timetable given to them from which they could pick their courses. A worthwhile project might be to try to formulate some method of generating fictitious students for the sectioning process that might include some way of guessing what options would be taken as well as the core courses that make up the various degree programs being examined.

A set of data was also used that was taken from the Computer Science (CPSC) department at the University of Calgary. This set of data proved to be easy to handle with the methods outlined in this thesis and, as such, does not really merit much discussion. The CPSC problem set, however, is smaller and less constrained than the Biology/Chemistry problem so this was not a surprise[8]. The CPSC data did not play a large role in the experimentation with either the Heuristic or the Genetic Algorithm approach. Whilst the set of CPSC data was useful in the early testing, the main focus of the research was on the more highly constrained Biology/Chemistry set of test data.

6.4 Genetic Algorithm Results

The Genetic Algorithm approach to solving the MTTP was not as successful as the heuristic approach. Some encouraging results were found however.

[7] If the list is very restrictive, they can be made into patterns in the **calendar.dat** file.

[8] For example CPSC laboratory sections can pretty much be placed in any room whilst the Biology and Chemistry lab sections almost always had to be held in certain specialized rooms. The CPSC courses also tended to have fewer laboratory sections with a shorter time duration (50 minutes per week compared with 170 or more minutes per week!).

Much experimentation must be undertaken when trying to fine tune a GA to a problem. Three types of crossover were employed in attempting to solve the problem. Parameters such as crossover rate and crossover type must be experimented with as well to try to find a combination that will give useful results consistently.

6.4.1 Crossover Types

Three types of crossover were used. They are:

1. Single point crossover.
2. Two-Point crossover.
3. Uniform crossover.

Of the three types of crossover tried it was two-point crossover that was found to be the most effective. Single point crossover came next followed by Uniform Crossover which performed the worst.

Looking again at how the solutions or chromosomes are represented could give some clues as to why this is the case. Each courses' lab/lecture/tutorial sections are placed together on the chromosome such that they are adjacent to one another. When single point or two-point crossover is carried out it is the case that, generally, courses' lab/lecture/tutorial sections are passed to the new chromosome unchanged or swapped with part of another solution (but at a maximum of two points for the entire chromosome in the case of two-point crossover). However, [BBM93b] states that "an advantage of having more crossover points is that the problem space may be searched more thoroughly". This suggests why two-point crossover works better than single point crossover.

It was found that uniform crossover performed worst of all. As mentioned above, [BBM93b] observes that increasing the number of crossover points means that more of the solution space is examined. The results show that this is probably the case but too many crossover points can be disruptive to solution formation as any "building blocks" (bits of a good schedule) are very likely to be broken up with uniform crossover. This can be explained by the "hashing" of schedules that uniform crossover would cause.

6.4.2 Parameters

Every time the GA is run, several parameters have to be specified. The program is run from the command prompt where the options are to be specified. They are:

1. Starting/Finishing crossover probability.
2. Starting/Finishing mutation probability.
3. Number of iterations.
4. Crossover type.

The crossover and mutation probabilities could either be kept constant throughout the application of the GA or could change over the runtime of the algorithm given a starting and finishing crossover value. These values would be supplied as a parameter. Generally the crossover was kept constant with the values tried (as outlined in the previous chapter) and the mutation probability was assigned an initial value of 0.002 and a final value of 0.05 with the mutation rate increasing in a linear fashion over the runtime of the algorithm.

Late in the GA process it is generally the case that the population begins to converge to similar or identical solutions. The mutation probability value is increased throughout the GA process to try to introduce more "new" genetic material in order to try to combat this process. Generally the GA is set to run 2,500 iterations with convergence occurring usually between the 1,500th and 2,000th iteration.

The population size of chromosomes is kept constant at 100. [Ree93] observes that the optimal size of the population for binary encoded strings grows exponentially with the length of the string. It seems reasonable to extend this to non binary strings by assuming a similar rate of growth would apply. Clearly this is not something which can be accommodated in practice. Generally population sizes of 50 or 100 are used in the literature with satisfactory results.

6.4.3 Prescheduling

It was found necessary to preschedule some of the sections. Courses that were targeted for prescheduling were ones that had a large number of laboratory instruction

type sections to schedule or were highly constrained full year courses with several laboratory sections. If there is a large number of these sections to schedule then there is often little choice in where they can be placed. Since there is little choice involved they can be scheduled manually and the rest of the schedule is generated "around them" taking into account any conflicts that the manually scheduled courses might cause.

When the prescheduling was not done, the GA found it difficult to find a useful solution; so prescheduling was deemed necessary from experience. The courses whose lectures and labs were prescheduled are BCEM441, BCEM443, BIOL231, BIOL311, CHEM201, CHEM203, CHEM350, CHEM515 and ECOL313. If the prescheduled course had any tutorial sections these were generally left to the GA to deal with as they usually only occurred once a week for about an hour so were fairly easy to schedule.

6.4.4 GA Statistics

As mentioned already, it was generally found that the results generated using two-point crossover were more effective than using single point or uniform crossover. Every solution generated has some conflicts. In the previous chapter the constraints that were used in the GA fitness function were discussed. A list of the average number of each type of conflict[9], as well as the average fitness, for 10 solutions generated from each crossover type with the parameters specified earlier is listed in Table 6.3.

Table 6.3 shows that 2-point crossover has the best performance amongst reduction of most types of conflicts and, hence on average, the best fitness also[10].

It is interesting to note that all 3 crossover methods have a average of 0 for conflict c_7. This is because this conflict is somewhat rare so it does not occur very often and if it does it is usually fairly easy for the GA to deal with (but has a high penalty attached to it as it deals with single section types).

[9]The constraint definitions can be found on page 58.

[10]Conflicts c_3 and c_6 are summed together because the conflicts are similar.

Constraint	2-Point	1-Point	Uniform
$v(c_1, x)$	0.6	0.8	0.5
$v(c_2, x)$	0.5	0.3	0.4
$v(c_3, x) + v(c_6, x)$	23.2	24.4	25.9
$v(c_4, x)$	0.2	0.4	0.6
$v(c_5, x)$	0	0	0.1
$v(c_7, x)$	0	0	0
$v(c_8, x)$	3.2	3.6	2.8
$v(c_9, x)$	6.3	8.0	10.9
$1000f(x)$	25.60	23.94	21.47

Table 6.3: Conflict Comparison between Crossover Types

6.4.5 Results

The results found vary in quality depending on the crossover type and parameter values used. The best results were found using 2-point crossover with a crossover probability of 0.20 and a changing mutation probability with an initial value of 0.002 and a final value of 0.05.

There are three types of solution that can come about from the application of the GA. They are:

1. Stand alone solutions.
2. Modifiable solutions.
3. Unusable solutions.

The first type of solution is the most desirable one. If the schedule produced has no room conflicts and shows no direct conflicts and a has a low or reasonable number of low/empty sections according to the sectioning program then the schedule produced is said to be "stand alone".

The second type of solution is also useful. It tends to have a small number of (small meaning up to three of four) room conflicts and/or direct conflicts. These conflicts can generally be fixed manually as they are usually laboratory sections that need to be moved or laboratory sections that have been scheduled at the same time

in the same room. With the minor modifications needed, it is easy to turn them into a stand alone solution.

The third type of solution is the worst result. It is generally a solution with many direct conflicts and possibly room conflicts. These conflicts are not always a result of many lecture sections having conflicting times with one another. They can also be more subtle conflicts in which a student's required courses over the fall and winter term have lecture sections that conflict with every available laboratory section of a course that is required[11]. They tend to be more serious in that the work required to fix them can be too great to be worthwhile. Examples of each will be given next.

In solutions of every type there are always some conflicts present. Recall from Section 5.1.2 that there are nine types of constraints taken into account by the fitness function. Each solution given by the GA also has an associated conflict file listing any constraints not honoured in the solution. The numbers of each type of conflict (or constraint violation) in the solution are recorded also.

Examples of each type of solution are given below.

Stand Alone Solution

A typical stand alone solution has a conflict array that is similar to the one shown in Figure 6.4. The GA groups c_3 and c_6 together as they are very similar constraints. The fitness function value is also multiplied by 1,000 so its value is greater than one although this is only for esthetic reasons when viewing the data.

As can be seen from Table 6.4 constraints c_1, c_4, c_5, c_7 do not appear in a stand alone solution. It is in fact the case that the following must hold for a solution, x, to potentially be classified as stand alone:

$$\sum_{i \in \{1,4,5,7\}} v(c_i, x) = 0 \tag{6.1}$$

This is a necessary, but not a sufficient, condition for a solution to be stand alone because, if there are any of those types of conflicts present in the schedule then it is impossible for a solution to be useful without any modification. The other conflicts

[11] This course with all the conflicting laboratory sections would, of course, have to be a full year course.

Constraint	Number of Appearances
$v(c_1, x)$	0
$v(c_2, x)$	0
$v(c_3, x) + v(c_6, x)$	25
$v(c_4, x)$	0
$v(c_5, x)$	0
$v(c_7, x)$	0
$v(c_8, x)$	2
$v(c_9, x)$	8
$1000f(x)$	26.52

Table 6.4: Conflict Example for Stand Alone Solutions

counted can be present but their numbers have dropped substantially from the initial solution. Initial solutions typically have a fitness value of between 2.5 and 3.5.

The solution represented in Table 6.4 has, as would be expected, no direct conflicts in the sectioning program. There are several empty/low enrollment sections however. These can be found in Table 6.5.

Course	Term/Inst Type/Secs	Enrollment(s)
BIOL311	F/B/6	17
CHEM341	F/B/3,5	1,1
CHEM350	B/B/8	0
CHEM350	B/T/4	0
CMMB343	F/B/2	0
CMMB443	W/B/2,4	0,0
ECOL313	F/B/1,6,11	0,0,0
ECOL317	W/B/4	0
ECOL419	W/B/2	0
ZOOL273	F/B/1,3,8	0,0,0
ZOOL461	F/B/3,4	0,0

Table 6.5: Typical Low Enrollment Sections from a Stand Alone Solution

If Table 6.5 is compared with the empty/low enrollments found in the actual timetable in Table 6.1 several things can be observed.

The first is that the timetable produced by the GA has six fewer sections with an

empty/low enrollment section than the real timetable. The second observation is that several courses that are listed as having an empty/low enrollment section in this GA result are not listed as having such in the real timetable. These courses are BIOL311, CMMB343, CMMB443, ECOL419, ZOOL273 and ZOOL461. These conflicts were investigated further using historical data to decide if these were serious conflicts or not.

For example consider the ZOOL461 empty enrollment lab sections. ZOOL461 has seven laboratory sections so on the surface this conflict would not seem to be very serious. With some data gathering from historical data and examination of which courses ZOOL461 conflict with it can be verified that this conflict is not very serious.

ZOOL461 B3 conflicts with the ECOL313 lecture section. ZOOL461 B4 conflicts with the BIOL311 lecture section. In the **calendar.dat** data file ZOOL461 is listed exactly once and is shown to be conflicting with BCEM443, ECOL313 and BIOL311. As the sectioning program only generates students with the courses as listed in the **calendar.dat** file it is easy to see why these sections are listed as empty.

In the historical data for the 93/94 session there were a total of 118 students listed as taking ZOOL461. Of these 41 of them were not taking any of BCEM443, ECOL313 or BIOL311 at the same time. Only 22 of the 118 students were listed as taking all three conflicting sections at the same time. It is easy to accommodate these students with the remaining laboratory sections of ZOOL461. Page 313 of the 93/94 calendar shows the same **calendar.dat** data for third year fall term Zoology students (which is where ZOOL461, BCEM443, ECOL313 and BIOL311 are constrained) as the 94/95 calendar so the comparison is valid [Cal93, Cal94].

Similar arguments can be made for all the other courses except for ECOL419 as it did not exist in the 93/94 session. Only one of its three laboratory sections has a conflict however so it should not have too much of a problem.

It is interesting to note that in some sense a harder problem is being solved. In practice most of the empty/low enrollment sections can easily be shown not to pose a problem for the schedules produced; this is done by looking at historical data and showing that the enrollment patterns do not pose a real problem for the schedules produced. This suggests that perhaps a relaxed set of constraints is needed. Unfortunately it would be very difficult to decide what those are without some close examination of historical data. Of course, as observed before, historical data is a

product of the schedule provided at the time. It is still a reasonable goal to try to reduce these conflicts however.

Several of the courses that have empty/low enrollment sections listed above actually have more empty sections than listed. This is due to the scheduling 'trick' outlined earlier in Section 4.2.6. BIOL311 has two laboratory sections for every one scheduled (since they share two rooms). CHEM350 laboratory sections can use four rooms (but one of them is shared with another course). Eight CHEM350 laboratory sections are scheduled but in actual fact 21 are required. Each CHEM350 empty/low enrollment section above represents two or three empty sections in the final schedule. In total there are 23 empty/low enrollment sections in this schedule. The actual schedule has 29 *but* it also has a direct conflict that this GA generated schedule does not. Generally most GA solutions have fewer than this number of empty/low enrollment conflicts so they can be thought of as slightly better than the actual schedule. Table 6.6 gives a conflict comparison between the actual 94/95 schedule and a typical stand alone GA result.

94/95 Schedule		GA Schedule	
Direct	Low/Empt Enroll	Direct	Low/Empt Enroll
1	29	0	14-30

Table 6.6: Conflict Comparison with 94/95 Schedule and a GA Schedule

Modifiable Solution

A solution that is considered to be modifiable is one that is useful, like the ones in the previous section, but requires some manual modifications to be made to it.

A sample modifiable schedule has a conflict array as shown in Table 6.7. As might be expected from that table there are some direct conflicts detected by the sectioning program. In this case there are three conflicts to be taken into account. ZOOL273 L1 and ECOL313 L1 conflict, CMMB241 L1 and CHEM354 L1 conflict and CHEM450 L1 and CHEM450 B3 conflict. As these conflicts involve single lecture section types they are unacceptable. For example, ZOOL273 and ECOL313 both only have one lecture section so it is impossible for any student to enroll in both which is clearly not allowed by definition of the **calendar.dat** file.

Constraint	Number of Appearances
$v(c_1, x)$	0
$v(c_2, x)$	2
$v(c_3, x) + v(c_6, x)$	16
$v(c_4, x)$	1
$v(c_5, x)$	0
$v(c_7, x)$	0
$v(c_8, x)$	4
$v(c_9, x)$	4
$1000f(x)$	25.83

Table 6.7: Conflict Example for Modifiable Solution

ECOL313 and ZOOL273 occur in the same recommended programme in the **calendar.dat** file four times. CMMB241 and CHEM354 occur in together once in the **calendar.dat** file also. The conflict between CHEM450 L1 and CHEM450 B3 should be self explanatory due to the fact that there is only one CHEM450 lecture section. This CHEM450 lecture section should not conflict with any of its associated laboratory sections. These three conflicts are violations of constraints c_2, c_2, and c_4 respectively.

Fortunately, with a bit of experimentation, it is possible to move these courses manually to remove these conflicts. When deciding which courses to move it often helps to examine the **calendar.dat** file to see which conflicting course is constrained by the most courses. The course that has the fewest number of constraints on it will probably be the easier of the two to move. Take the conflict between ECOL313 and ZOOL273 for example. If the **calendar.dat** file is examined the courses having constraints involving ECOL313 as well as ZOOL273 (individually) can be found in Table 6.8.

As can be seen from Table 6.8 it is ECOL313 that is the most constrained. Table 6.8 also shows the timeslots that each course's lecture is assigned. ZOOL273 is constrained by fewer courses and, therefore, is a probable candidate to move. By examining the times at which courses having constraints with ZOOL273 are scheduled there are several candidate times that ZOOL273 can be moved into. Unfortunately

ZOOL 273	MWF 1700 50	ECOL 313	MWF 1700 50
BIOL 311	MWF 1000 50	BCEM 443	MWF 1200 50
BOTA 323	MWF 1200 50	BIOL 311	MWF 1000 50
CHEM 341	MWF 1300 50	BOTA 225	MWF 1100 50
CHEM 350	MWF 0800 50 TR 1700 75	BOTA 323	MWF 1200 50
CHEM 354	TR 0800 75	CHEM 341	MWF 1300 50
CHEM 410	TR 1530 75	CHEM 350	MWF 0800 50 TR 1700 75
ECOL 313	*MWF 1700 50*	CHEM 354	TR 0800 75
		ZOOL 273	*MWF 1700 50*
		ZOOL 461	MWF 1400 50

Table 6.8: Calendar Conflicts with ZOOL 273 and ECOL 313

one also must consider the times that the laboratory sections (all 10 of them) for this course are scheduled. TR at 1100 for 75 minutes is a legal time with all the parameters taken into consideration (including room conflicts). By moving ZOOL273 to this time the direct conflict has been removed from the schedule. This is the simplest case. The conflict between CMMB241 and CHEM354 is more difficult to manually resolve.

It should be obvious that CMMB241 is the candidate course to be moved. Since CHEM354 is a *full year* course moving it could have drastic consequences as it can potentially conflict with many more courses than would a fall/winter term course. Unfortunately, in this particular schedule, due to the list of sections potentially conflicting with CMMB241 and where the laboratory sections of CMMB241 are placed there is no alternative non-conflicting timeslot in which to place CMMB241. One of the sections that is constrained by CMMB241 (as a calendar conflict) has to be moved so CMMB241 can take its place. Fortunately BIOL315, which is constrained by CMMB241, has a relatively small list of courses it is constrained by so it is an ideal candidate for this strategy. BIOL315 is on MWF 1300 for 50 minutes. It will be moved to MWF 1100 for 50 minutes. This makes the MWF 1300 timeslot available for CMMB241. The next step is to find a free room for each of these two sections at the new times (which is easily done in this case) and this conflict has been successfully

removed from the schedule.

The final conflict to move is the CHEM410 B3 section that conflicts with its associated lecture section. Fortunately, in this case, it is easy to accommodate. This laboratory section takes place at T 1800 for 230 minutes. This conflicts with the CHEM410 lecture section which takes place on TR at 1700 for 75 minutes. Moving the laboratory section to Wednesday removes this conflict.

Fixing the conflicts manually can be relatively straightforward or require a bit more thought (as shown with the CMMB241 conflict with CHEM354). However after these modifications were made the schedule is now a "stand alone" solution that can be used as is. Generally the number of empty or low enrollment sections is similar to the stand alone solutions. It should be noted, however, that when courses are moved about it can be the case that more empty or low enrollment sections are created as a result.

Unusable Solution

This is a class of solutions that are harder to turn into useful solutions though modification. The few schedules that fall into this category generally have a low fitness value and many conflicts. When applied to the sectioning program there is a larger list of direct conflicts that are observed. Generally this conflict list will include *laboratory sections* in addition to lecture sections. Table 6.9 shows an example of what the conflicts would be for an unusable solution.

As can be seen from this table this particular schedule has a below average fitness. A list of direct conflicts for this schedule can be found in Table 6.10. In most cases it would take a substantial effort to convert one of these types of solution into a usable one. Much of the work is contained in identifying why the direct conflicts exists. The example outlined here could be considered one of the easier cases (which is still difficult).

As it turns out there are two serious conflicts in this schedule (in addition to two room conflicts). The schedule whose conflicts are listed in Table 6.10 has a problem that relates to entries in the **calendar.dat** file. The troublesome part of the **calendar.dat** file is the course programmes listed below:

Constraint	Number of Appearances
$v(c_1, x)$	2
$v(c_2, x)$	0
$v(c_3, x) + v(c_6, x)$	24
$v(c_4, x)$	0
$v(c_5, x)$	0
$v(c_7, x)$	0
$v(c_8, x)$	4
$v(c_9, x)$	3
$1000f(x)$	23.04

Table 6.9: Conflict Example for an Unusable Solution

Course	Term/Inst Type/Secs
BOTA323	F/L/1
CMMB241	W/L/1
CHEM350	B/B/1,2,3,4,5,6,7,8
CHEM354	B/B/1,2,3
CMMB301	W/L/1
BIOL311	F/L/1
BIOL315	W/L/1
BOTA327	W/L/1
BOTA327	W/B/1

Table 6.10: Direct Conflicts Example from an Unusable Solution

```
C 2nd year/1st term botany
P BOTA323  CHEM350  ZOOL273  ECOL313

C 2nd year/2nd term botany
P BOTA327  CHEM350  CMMB241  CMMB301

C 2nd year/1st term ecology
P BIOL311   BOTA225   CHEM341   ECOL313
P BIOL311   BOTA225   CHEM350   ECOL313
P BIOL311   BOTA225   CHEM354   ECOL313
P BIOL311   ZOOL273   CHEM341   ECOL313
P BIOL311   ZOOL273   CHEM350   ECOL313
```

```
P BIOL311    ZOOL273    CHEM354    ECOL313

C 2nd year/2nd term ecology
P BIOL315    CMMB241    CHEM350    ECOL317
P BIOL315    CMMB241    CHEM354    ECOL317
P BIOL315    CMMB241    CHEM341    ECOL317
```

The first serious direct conflict can be identified from the fact that *all* of the CHEM350 laboratory sections appear in the direct conflict list. Examination of the programme enrollment recommendation for second year first and second term Botany shows where the conflict occurs, however. The sectioning program generates students that are taking courses over the fall *and* winter terms. Since CHEM350 is a full year course the students enrolled in second year Botany would have CHEM350 listed as a course taken in both the fall and winter terms. Unfortunately, because BOTA323, BOTA327, CMMB241 and CMMB301 together conflict with *all* the CHEM350 laboratory sections it is impossible for a student to enroll in this programme of study. A clue to the second direct conflict can be found in the fact that all the CHEM354 laboratory sections also appear in the direct conflict list. Similar to the CHEM350 case above an examination of the **calendar.dat** file reveals the source of this conflict. Second year Ecology recommended courses BIOL311 and BIOL315 conflict with *all* of the CHEM354 laboratory sections. It is impossible for a student to enroll in all of the recommended programmes for second year Ecology because of this. There are many recommended patterns of enrollment for second year Ecology and most of them can be satisfied with the current schedule but it *must* be the case that they can all be satisfied.

An approach to solve this could be to move some of the CHEM350 and CHEM354 laboratory sections to non-conflicting times. This can be difficult because CHEM350 and CHEM354 are both *full year* courses and they are difficult to move as it would have a direct impact on courses in both the Fall and Winter terms. It does not help the situation either that both CHEM350 and CHEM354 occur very frequently in recommended enrollment patterns in the **calendar.dat** file. Alternatively, some of the conflicting lecture or lab sections could also be moved to try to remove one of the conflicts with the CHEM350 and CHEM354 laboratory sections.

Unfortunately this schedule also has two room conflicts. Generally, unusable

schedules have several room conflicts. These can be difficult to accommodate if the room is already being used by many courses in the schedule. If the courses involved in the room conflict are also highly constrained (conflict with many courses) that makes it more difficult to modify the solution. Most room conflicts, as is the case with the schedule being discussed here, occur between laboratory sections of the same course trying to use the same room at the same, or an overlapping, time. If there are several of these sections, modifications can prove difficult as it can sometimes be the case that the GA has built up the rest of the potentially conflicting courses around these laboratory sections in their overlapping times. Laboratory sections are generally around three hours in length so moving a highly constrained laboratory section without introducing more empty or low enrollment sections is difficult. The more that the schedule has to be modified the more likely it is that more conflicts may be introduced into it.

It is more difficult to modify a solution, such as this one, to be useful than a solution that is classified as "modifiable". The example given above is one of the easier cases found in the set of unusable solutions. The worst solutions tend to have well over 20 direct conflicts and several room conflicts. Unusable solutions can also have the property that they have too many empty or low enrollment sections to be practical. If the vast majority of a course's laboratory sections have an empty/low enrollment then the schedule will probably not be very useful in practice and it could be quite difficult to move these sections around to change this fact. The work involved to fix such a problem becomes more difficult in such situations and, because of this, these schedules are generally classified as "unusable" both due to the number of direct conflicts present as well as the additional problem of several room conflicts to fix.

6.5 Heuristic Results

The heuristic algorithm provided the better results of the two methods tried. Runtime is typically a few seconds but due to its probabilistic nature it is possible that it can make a poor choice and take up to ten minutes. Regardless, the results produced by it are most encouraging.

A typical result from the heuristic approach generates a solution with *no* direct

conflicts. The number of sections that have a low or empty enrollment according to the sectioning program is generally much lower than what is found in the actual schedule. Typically between two and fifteen such sections will exist. Since the heuristic algorithm contains probabilistic choices the solutions generated are usually somewhat different from one another.

A typical set of low/empty enrollment courses identified by the sectioning program with a heuristic timetable is given in Table 6.11. As can be seen from the table there are far fewer empty/low enrollment sections than in both the actual solution and the GA solutions. There are also no direct conflicts.

Course	Term/Inst Type/Secs	Enrollment(s)
BOTA225	F/B/3,7	3, 5
CMMB241	W/B/7	0
ZOOL273	F/B/3,8	1,3

Table 6.11: Typical Low Enrollment Sections from a Heuristic Solution

It is of interest that CMMB241 laboratory sections do not appear in the empty/low enrollment section list of the actual 94/95 timetable. ECOL317 and CMMB301 lecture sections were identified as the conflicting sections with the CMMB241 low enrollment section in this particular schedule. This was examined in more detail with a look at the enrollment patterns of the 93/94 academic year. In this year there were 336 students enrolled in CMMB241. 68 of these students were enrolled in CMMB301 in addition to CMMB241 and 10 were enrolled in both ECOL317 and CMMB241. Only two students were enrolled in all three courses. This means that the 76 students that can't use this CMMB241 laboratory section have to choose from the nine other laboratory sections outside of the unavailable one which should not pose any real problem. A similar situation arises with the BOTA225 low enrollment sections. They both conflict with CHEM350 and CHEM341 lecture sections and the first laboratory section listed also conflicts with a CHEM354 laboratory section. Fortunately out of the 134 students that were enrolled in BOTA225 in the 93/94 session only 27 were enrolled in CHEM350 at the same time as were 14 and 2 students enrolled in CHEM341 and CHEM354 respectively at the same time. This conflict with BOTA225 should also cause little problem. The conflict with ZOOL273 laboratory sections can be ex-

plained in a similar manner (it should also be observed that the actual 94/95 schedule has empty ZOOL273 sections whilst the ones contained in the heuristic schedule in question are not quite empty).

Similar to the CHEM350 empty sections that are in the actual schedule, the CMMB241 empty section in this schedule is due to the restrictions found in the **calendar.dat** file. The courses conflicting with the CMMB241 laboratory section in Table 6.11 appear at least once in any line in the **calendar.dat** file that also contains CMMB241. Courses that have a large number of laboratory instruction type sections generally do not have conflicts with them treated as strict conflicts in the heuristic. This is why there are a few of them that appear as low enrollment sections in the sectioning program.

As mentioned above, the results generated by the heuristic generally have *no* direct conflicts and have typically between two and fifteen sections with an empty or a low enrollment. This number includes the fact that some of the laboratory sections will be *duplicated* as mentioned earlier in Section 4.2.6. Table 6.12 gives a conflict comparison between the actual 94/95 schedule and a typical heuristic result. At this point it is interesting to note the importance of the probabilistic conflict with regards to the quality of the final solution. If this conflict is ignored then solutions are generated with virtually no backtracking almost all of the time. The difference is there are far too many empty/low enrollment sections for the schedule to be practical. A schedule generated in this fashion can be expected to have well over 25 or 30 empty/low enrollment sections. This demonstrates how important the probabilistic conflict check is to the success of the heuristic algorithm.

94/95 Schedule		Heuristic Schedule	
Direct	Low/Empt Enroll	Direct	Low/Empt Enroll
1	29	0	2-15

Table 6.12: Conflict Comparison with 94/95 Schedule and Heuristic Schedule

Theoretically, conflicts such as those found in the unusable GA solutions could be found in a heuristic result. This is due to the fact that, as in the GA, conflicts over both the fall and winter sessions are not considered. This would seem to occur very rarely if at all. Solutions can generally be generated quickly and reliably with the

heuristic. Also, it should be noted, the heuristic never has any room conflicts.

6.6 Discussion

This thesis has been about the application of two methods to the MTTP. A custom heuristic approach was developed as well as a Genetic Algorithm. Both methods examined do produce results that are generally better than the actual schedule. There are still more questions to be asked of the results that perhaps future research can answer. A closer look at how fictitious students are generated may be called for to try to make the fictitious student generation more "realistic".

The heuristic approach produced the best solutions of all. Each solution generated will be different from the previous one due to the probabilistic nature of the algorithm making it useful in generating different results although it is the case that some sections will be scheduled in similar positions each time the algorithm is applied. Fortunately with the flexibility of defining timeslot classes the user could, if desired, restrict various course instruction types to certain parts of the day if it was found to be needed. The heuristic can generate good schedules quickly and reliably.

The heuristically generated solutions contained far fewer conflicts than the actual timetable used as well as the GA produced schedules.

The GA approach was somewhat less satisfactory than the heuristic. It was slower (hours instead of minutes) and produced solutions that were not as good as the heuristic. It was also not reliable in the sense that it did not produce good "stand alone" solutions most of the time. The GA method, however, did produce different solutions with each run. There is a possible explanation for the lack of performance from the GA however. *Epistasis* is defined in [BBM93b] as:

> "The term *epistasis* has been defined by geneticists as meaning that the influence of a gene on the fitness of an individual depends on what gene values are present elsewhere."

Clearly the master timetable problem falls into this category. When a gene (section) has its room or timeslot changed its effect on the fitness of the chromosome is dependent on the values of the other genes. The modification could improve the solution or

make it worse but this is decided by the value of the other genes present in the solution. The penalty based fitness function reflects this by assigning a lower fitness value to a chromosome for each pair of conflicting genes. According to the *building block principle* it is short low order schemas that are necessary requirements for a good solution to be found. Since the building block's fitness, in the case of scheduling, is directly related to the rest of the chromosome it is easy to see where the building block principle may not hold and why it is that GA's do not work in situations like this as well as they do in other situations [BBM93b].

GA solutions are classified into three types. They are:

1. Stand alone solutions.
2. Modifiable solutions.
3. Unusable solutions.

As the names imply the first type is the most desirable as it can be applied with no modification. The second type of schedule generated requires some manual modifications to make it feasible (due to room conflicts or course conflicts). The third type has so many serious conflicts that it is difficult to modify and should be discarded.

It was found that two-point crossover with a crossover probability of 0.20 and a linearly increasing mutation probability starting at 0.002 and finishing at 0.05 were the most effective parameter settings used. Table 6.13 gives a listing of the distribution of the different types of solutions generated using these parameters. This was taken from 10 sample solutions generated.

Solution Type	2-Point
Stand Alone	3
Modifiable	4
Unusable	3

Table 6.13: Distribution of Solution type according to Crossover Method

There are several factors that are not considered in either approach at the moment. The rest of the courses that make up a degree programme are not considered in much detail here (for example, options) as they generally have much more flexibility than

the required courses as outlined by the calendar enrollment recommendations. This could be a topic for future research as it is related to developing a method of creating fictitious students for the sectioning process.

The type of direct conflicts that occur in unusable GA solutions (and, possibly, in the heuristic solution) is often a result of the GA not counting conflicts over the entire year. The conflicts counted are only over the fall and winter terms individually. It would be worthwhile to investigate how these conflicts could be accounted for but it was found that they were uncommon enough to not really represent a problem. The heuristic also does not take conflicts into account over the full year but it is rarely a problem with this approach.

Chapter 7

Conclusion

This thesis has examined two methods to solve the master timetable problem. The first method examined was a heuristic algorithm; the second method applied to the MTTP was a Genetic Algorithm. The Biology and Chemistry timetable data from the University of Calgary was used for testing purposes as it is a particularly challenging data set. The methods are somewhat generic and, with some modification, could be used for other timetabling data as well[1].

The heuristic approach tried to solve the problem in an incremental manner. By assigning timeslots and rooms to a single course section at a time a solution is gradually built up. Backtracking was applied to reverse choices made earlier in the algorithm in order to try the assignment of different values to some courses when the current state is found to be infeasible. This approach works well but may need some slight modification in the conflict routines to reflect what is defined as a serious conflict if different data is used. The heuristic approach discussed here produces a different schedule each time it is run due to the probabilistic choices made. If some schedules are deemed unfit, for various reasons, a new schedule can easily be generated that might be more suitable. This can be accomplished through modification of the conflict routines, modification of the data files[2] or by re-running the heuristic as it will generate a new solution each time it is invoked.

The genetic algorithm approach uses a vastly different line of reasoning. A popu-

[1] It was used with the CPSC data but that problem set is considerably easier than the Biology/Chemistry dataset.

[2] To restrict undesirable timeslots or rooms if desired.

lation of chromosomes (solutions) is initially generated. Each iteration of the genetic algorithm uses techniques such as crossover and mutation to merge or modify previous solutions in order to generate new solutions. The reasoning behind this is that if one or both of the solutions contain some inherently good characteristics, then it is hoped that the offspring of these two solutions will combine the good characteristics of both to create a stronger solution. Mutation tries to introduce more genetic diversity into the solutions. Over many generations, better and better solutions are generated. One advantage of the genetic algorithm method is that it is generic in the sense that little has to be changed in the algorithm itself to enable it to be applied to different situations and problems. To enable the GA to be productive with a new set of data a new fitness function would need to be developed to identify good schedules from poor ones. The rest of the internal workings of the GA could be left intact. Unfortunately, experimentation must be carried out in order to fine tune a new fitness function such that it can identify the quality of solutions. Experimentation would also be needed in order to identify reasonable values for parameters such as crossover probability. The runtime of a GA is also typically much longer than that of a heuristic. Like the heuristic approach the GA approach generates different schedules each time it is applied.

Comparing between the two it is the heuristic that seems to be the better choice. Although it is more tricky to modify from one scheduling scenario to another it gives better results faster. The Genetic Algorithm is too slow and its results do not compare well with those of the heuristic algorithm. The methods employed here, especially the heuristic, did meet the goals set out at the outset of this thesis.

The suite of programs that have been a product of this research could possibly be packaged and made into a generally usable scheduling system. This would involve a considerable amount of work as, for example, no user friendly interface has been designed for it as both methods are strictly "command line" interface programs.

7.1 Future Work

There is more that can be done with this research in an attempt to make it more useful to generic timetabling in general. One area of the timetabling problem that

was not part of this thesis was the scheduling of optional courses in a degree programme. The core courses to be taken with the degree programmes offered by the Biology/Chemistry department are catered for here but nothing has been done with respect to how options could be included in the scheduling.

Many of the problems with options are due to the historical patterns resulting from choices made in the context of an existing timetable. Optional courses generally are up to the students to take at their discretion. There are also required courses that are to be taken as part of a degree programme but it is up to the student when to take them (many 3rd and 4th year courses fall into this category). What courses these should and should not be permitted to conflict with is not entirely clear. Examination of historical data is useful to see what enrollment patterns are present over the years but, unfortunately, these patterns are a result of the timetable itself, from which the students had to choose their courses. A method that encapsulates both historical information as well as some idealistic information would probably be the best suited to tackle this and would be an interesting project.

The program used for evaluating the schedules has some room for future improvements. Currently it generates fictitious students based entirely on the course patterns contained in the **calendar.dat** file. Optional courses and required courses not listed in the **calendar.dat** file should be included in schedule evaluation so this is also an issue that could be the topic of a future project.

The methods examined here have been tested on one of the most difficult cases within the University of Calgary and they have done quite well with this case. Some work would have to be undertaken to extend their range of application to include other courses from departments outside of Biology and Chemistry (for example, from departments such as Mathematics or Physics).

The two methods presented in this thesis did produce successful timetables. With the inherent difficulties with the data used this is a good achievement and would make an ideal platform to tackle larger scheduling problems in the future.

Bibliography

[BBM93a] D. Beasley, D. R. Bull, and R. R. Martin. An overview of genetic algorithms: Part 1, fundamentals. *Univeristy Computing*, 15(2):58–69, 1993.

[BBM93b] D. Beasley, D. R. Bull, and R. R. Martin. An overview of genetic algorithms: Part 2, research topics. *Univeristy Computing*, 15(4):170–181, 1993.

[Bre79] D. Brelaz. New methods to colour the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

[Cal93] *University of Calgary 1993-1994 Calendar*, 1993.

[Cal94] *Univeristy of Calgary 1994-1995 Calendar*, 1994.

[Cam93] Neil A. Campbell. *Biology*. Benjamin/Cummings, 1993.

[Cam94] Peter J. Cameron. *Combinatorics: Topics, Techniques, Algorithms*. Cambridge University Press, 1994.

[CBBC+95] F. P. Coenen, B. Beattie, T. J. M. Bench-Capon, M. J. R. Shave, and B. M. Diaz. Spatial reasoning for timetabling: The timetabler system. In *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, pages 57–68, 1995.

[CHdW87] M. Chams, A. Hertz, and D. de Werra. Some experiments with simulated annealing for colouring graphs. *European Journal of Operational Research*, 32:260–266, 1987.

[CKLW95] Czarina Cheng, Le Kang, Norrus Leung, and George M. White. Investigations of a constraint logic programming approach to university timetabling. In *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, pages 82–93, 1995.

[CL95a] Anton W. Colijn and Colin J. Layfield. Conflict reduction in examination schedules. In *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, pages 82–93, 1995.

[CL95b] Anton W. Colijn and Colin J. Layfield. Interactive improvement of examination schedules. In *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, pages 297–306, 1995.

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

[Col71] Anton W. Colijn. Reduction of second-order conflicts in examination timetables. Technical report, Computer Science Department, University of Waterloo and University of Calgary, 1971.

[Col73] Anton W. Colijn. A sectioning algorithm. *INFOR*, 11(3), October 1973.

[CR95] Dave Corne and Peter Ross. Peckish initialization strategies for evolutionary timetabling. In *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, pages 69–81, 1995.

[CRF94a] Dave Corne, Peter Ross, and Hsiao-Lan Fang. Evolutionary timetabling: Practice, prospects and work in progress. Presented at the UK Planning and Scheduling SIG Workshop, Strathclyde, September 1994.

[CRF94b] Dave Corne, Peter Ross, and Hsiao-Lan Fang. Fast practical evolutionary timetabling. In *Proceedings of the AISB Workshop on Evolutionary Timetabling.* Springer-Verlag, 1994.

[Dav91] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[DeW71a] D. DeWerra. Balanced schedules. *INFOR*, 9(3), November 1971.

[DeW71b] D. DeWerra. Construction of school timetables by flow methods. *INFOR*, 9(1), February 1971.

[Fan92] Hsiao-Lan Fang. Investigating genetic algorithms for scheduling. Master's thesis, University of Edinburgh, 1992.

[Gol89] David E. Goldberg. Zen and the art of genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 80–85, 1989.

[Got63] C. C. Gotlieb. The construction of class-teacher timetables. In *Proc. IFIP Congress*, pages 73–77. North-Holland Publishing Company, Amsterdam, 1963.

[Hol75] John H. Holland. *Adaptation in Natural and Artificial Systems.* Massachusetts Institute of Technology Press, 1975. First edition University of Michigan Press.

[ICP95] *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, 1995.

[KB95] Ahamad Tajudin Khader and John T. Buchanan. School timetabling: A knowledge-based approach. In *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, pages 99–111, 1995.

[KDS89] A. Kenneth, DeJong, and William M. Spears. Using genetic algorithms to solve np-complete problems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 124–132, 1989.

[KM95] A. C. Kakas and A. Micheal. Timetabling in an integrated abudctive and constraint logic programming framework. In *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, pages 167–176, 1995.

[Lay93] Colin Layfield. Registrar final examination timetabling system. Technical report, Computer Science Department, University of Calgary, 1993.

[LK73] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the travelling salesman problem. *Operations Research*, 1973.

[Man81] Bennet Manvel. Colouring large graphs. *Congressus Numerantium*, 33:197–204, December 1981.

[Mic92] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, 1992.

[PD95] Charles C. Peck and Atam P. Dhawan. Biochemical techniques take on combinatorial problems. *Dr. Dobbs Journal*, August 1995.

[Pee83] Jürgen Peemöller. A correction to brelaz's modification of brown's colouring algorithm. *Communications of the ACM*, 26(8):595–597, 1983.

[PVTF92] Press, Vetterling, Teukolsky, and Flannery, editors. *Numerical Recipies in C.* Cambridge University Press, 1992.

[Ran95] R. C. Rankin. Memetic timetabling in practice. In *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, pages 45–56, 1995.

[RC94] Peter Ross and Dave Corne. Applications of genetic algorithms. Technical Report 94-007, University of Edinburgh, Department of Artificial Intelligence, Genetic Algorithms Research Group, 1994.

[RCF94a] Peter Ross, Dave Corne, and Hsiao-Lan Fang. Improved evolutionary timetabling with delta evaluation and directed mutation. In Y. Davidor and H-P. Schwefel, editors, *Parallel Problem Solving from Nature III.* Springer-Verlag, 1994.

[RCF94b] Peter Ross, Dave Corne, and Hsiao-Lan Fang. Successful lecture timetabling with evolutionary algorithms. In *Proceedings of the AISB Workshop on Evolutionary Timetabling.* Springer-Verlag, 1994.

[Ree93] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* Wiley, 1993.

[Ric95] David C. Rich. A smart genetic algorithm for university timetabling. In *Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, pages 202–216, 1995.

[Smi95] Barbara M. Smith. A tutorial on constraint programming. Technical Report 95.14, School of Computer Studies, University of Leeds, 1995.

[SP94] M Srinivas and M. Patnaik. Genetic algorithms: A survey. *Computer*, June 1994.

[Sti94] David Stirzaker. *Elementary Probability.* Cambridge University Press, 1994.

[Sys89] G. Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–8, 1989.

[Tan89] Andrew Tanenbaum. *Computer Networks.* Prentice-Hall, 1989.

[Tuc84] Alan Tucker. *Applied Combinatorics.* Wiley, 1984.

Appendix A

Datafiles Used

A.1 The calendar.dat file.

The **calendar.dat** file contains information on what restrictions are to be placed on the timetable. Each term in each year in most programmes of study at the University of Calgary generally has a list of courses that are taken that form the basis of the degree (the required course component). An example of a set of such restrictions in the file should provide enough detail to understand what information it supplies.

Take the following section of the **calendar.dat** file for example:

```
N F 5
C 2nd year/1st term honors biochemistry
P BIOL311   BOTA225   CHEM354   CHEM410
P BIOL311   ZOOL273   CHEM354   CHEM410
```

A line beginning with the character 'N' describes a new programme of study. In this case it is the 2nd year/1st term honours biochemistry. The first character after the 'N' is typically 'F' or 'W' to represent the term it takes place in (Fall or Winter). The next value is an integer representing the expected enrollment in this programme. Lines starting with a 'C' character are comments (in this case describing what programme it is). This information is used by the sectioning program to help determine how many fictitious students to generate as well as what courses they will be enrolled in.

Lines starting with a 'P' character show a possible way which a student in this programme may enroll. In the example above the two lines (starting with a 'P' character) show two possible ways a student in this programme could enroll. The calendar conflict matrix is built based on this information. Each course in a given 'P' row will be marked as conflicting with one another. These conflicts are reflected in the calendar conflict matrix.

As mentioned above, the enrollment information given for each programme type is used in the sectioning program that tests the schedules generated. The number of students enrolled in a particular programme option is usually biased by how many students in the historical data are found to be enrolled in the same programme. The students generated will typically have one of the sets of courses listed in the programme they are to belong to.

A partial listing of the **calendar.dat** file for the 1994-1995 calendar year can be found in Section A.1.1.

A.1.1 Partial listing of the calendar.dat file used.

```
C       Calendar data for 1994 - 1995
C DEPARTMENT OF BIOLOGY
C
N F 75
C  1st year/1st term biochemistry
P BIOL231   CHEM201   MATH251   PHYS201
P BIOL231   CHEM201   MATH251   PHYS231
N W 75
C 1st year/2nd term biochemistry
P BIOL233   CHEM203   MATH211   PHYS203
P BIOL233   CHEM203   MATH211   PHYS233
P BIOL233   CHEM203   MATH253   PHYS203
P BIOL233   CHEM203   MATH253   PHYS233
N F 5
C 2nd year/1st term honors biochemistry
P BIOL311   BOTA225   CHEM354   CHEM410
P       BIOL311   ZOOL273   CHEM354   CHEM410
N F 75
C 2nd year/1st term normal biochemistry
P BIOL311   BOTA225   CHEM350   CHEM410
P BIOL311   BOTA225   CHEM354   CHEM410
P       BIOL311   ZOOL273   CHEM350   CHEM410
P       BIOL311   ZOOL273   CHEM354   CHEM410
N W 5
C 2nd year/2nd term honors biochem
P BCEM441   CMMB301   CHEM354   CHEM410
N W 75
C 2nd year/2nd term normal biochem
P BCEM441   CMMB301   CHEM350   CHEM410
P BCEM441   CMMB301   CHEM354   CHEM410
N F 80
C 3rd year/1st term biochem
P BCEM443   BCEM471   BCEM531
N W 80
C 3rd year/2nd term biochem
P BCEM473   BCEM547
```

```
N F 80
C 4th year/1st term biochem
P BCEM541   BCEM537   BCEM530
N W 80
C 4th year/2nd term biochem
P BCEM551   BCEM530
```

A.2 The combo.dat file.

The **combo.dat** file lists the courses that are to be scheduled. It gives vital information such as how many sections of each instruction type are to be scheduled for each course. Information on the room and timeslot categories is also given.

The file used can be found below. An explanation of the file format is given also.

A.2.1 Partial listing of the combo.dat file used.

```
C=============================================================
C FALL 94 / WINTER 95 timetable for BIO and CHEM
C
C BCEM341   1209 1 01 00 2     A 0090 LEESS01  T001 R000 150
C \__/\_/   \__/ | \/ \/ |     | \__/  |       \__/ \__/ \_/
C  A   B     C   D E  F  G     H  I    J        K    L    M
C 012345678901234567890123456789012345678901234567890123456789
C
C  A: course name.  B: course number.  C: department id.
C  D: Section Instruction Type  E: Number of sections
C  F: Unused  G: Term  H: Unused  I: Enrollment Capacity
C  J: Instructor (If known)  K: Timeslot Category
C  L: Roomslot Category  M: Minutes per week
C=============================================================
C BCEM courses in calendar programs (16)
C
C 114 course combos total.  FALL 94 / WINTER 95 timetable
C for BIO and CHEM
C
C BCEM courses in calendar programs (16)
C
A BCEM341   1209 1 01 00 2     A 0090 LEESS01  T001 R000 150
A BCEM341   1209 2 04 00 2     A 0090 OLSOB01  T002 R001 170
A BCEM441   1209 1 01 00 2     A 0320 VOORG01  T001 R002 150
A BCEM441   1209 2 06 00 2     A 0320 OLSOB01  T003 R003 230
C BCEM441 has labs on alternating weeks,
C so the actual number of sections is 12.
A BCEM443   1209 1 01 00 1     A 0160 HUBERE01 T001 R004 150
A BCEM443   1209 2 09 00 1     A 0136          T003 R001 230
A BCEM471   1209 1 01 00 1     D 0473 HUBERE01 T001 R000 150
```

```
A BCEM471   1209 3 01 00 1     D 0135 HUBERE01 T004 R007 110
A BCEM473   1209 1 01 00 2     D 0100 MCIND01  T001 R006 150
A BCEM473   1209 3 01 00 2     D 0120 VOORG01  T004 R007 110
A BCEM531   1209 1 01 00 1     D 0100 HUBERE01 T001 R006 150
A BCEM537   1209 1 01 00 1     D 0050 VANDJH02 T001 R007 150
A BCEM541   1209 1 01 00 1     D 0040 VOORG01  T001 R008 100
A BCEM541   1209 2 03 00 1     D 0040 OLSOB01  T014 R001 350
C this class has a tutorial scheduled
C on the third day of the lectures.
A BCEM547   1209 1 01 00 2     D 0075 GAUCGM01 T001 R000 150
A BCEM551   1209 1 01 00 2     D 0060 HILLBC01   T001 R008 150
A BCEM553   1209 1 01 00 2     D 0065          T001 R007 150
A BCEM555   1209 1 01 00 1     D 0040          T001 R017 150
```

A.3 The rooms.dat file.

The **rooms.dat** file contains information on the rooms used in the scheduling process. Rooms are listed by *room category* with each category consisting of a list of rooms that belong to it.

Information given on an individual room consists of the name of the room as well as its capacity.

A partial listing of the **rooms.dat** file is given next.

A.3.1 Partial listing of the rooms.dat file used

```
C Roomslot category 0
0 0
C Classrooms with capacity around 90.
R SS 0109  0077 0000
R SS 0113  0075 0000
R ST 0133  0090 0000
R ST 0139  0090 0000
R ST 0141  0125 0000
R ST 0145  0125 0000
C
C Roomslot category 1
0 1
C biochem lab for BCEM341/BCEM443
R BI 0138  0018 0000
C
C Roomslot category 2
0 2
C the BIG science theaters.
R ST 0140  0408 0000
R ST 0148  0408 0000
```

A.4 The ts.dat file.

The **ts.dat** file contains information on the timeslots used in the scheduling process. Timeslots, similar to the **rooms.dat** file, are listed by *timeslot category* with each category consisting of a list of timeslots that belong to it.

Information supplied about an individual timeslot consists of:

1. The days of the week the timeslot is active.
2. The time on the above days the timeslot starts.
3. The duration of the timeslot.

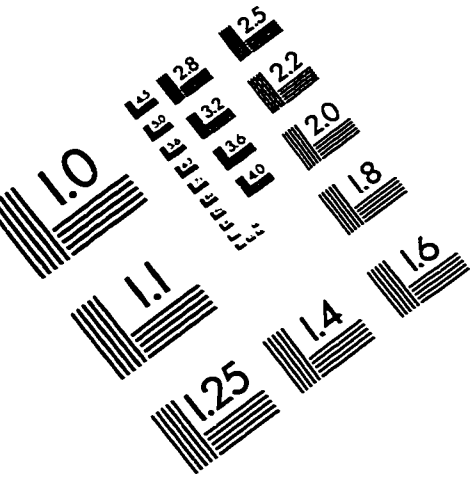
A partial listing of the **ts.dat** file is given next.

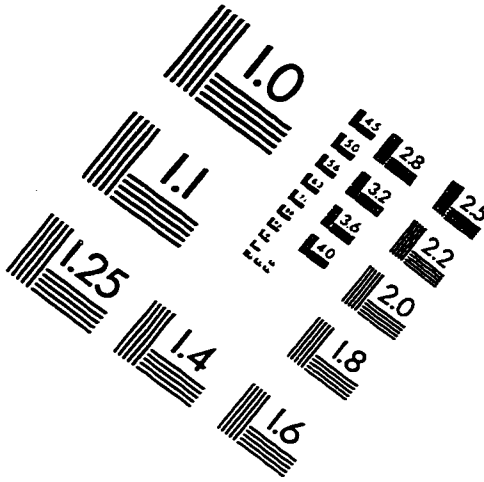
A.4.1 Partial listing of the ts.dat file used

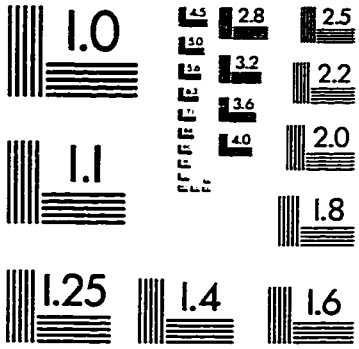
```
.

C Timeslot category 0
N       0
C Standard lecture A+B (8-17)
T       MWF     1200    50
T       MWF     1400    50
T       MWF     1100    50
T       MWF     1300    50
T       MWF     1000    50
T       MWF     1500    50
T       MWF     900     50
T       MWF     1600    50
T       MWF     800     50
T       MWF     1700    50
T       TR      1230    75
T       TR      1530    75
T       TR      930     75
T       TR      1400    75
T       TR      1100    75
T       TR      800     75
T       TR      1700    75
C
C Timeslot category 11
N       11
C 4 hour labs for chemistry (require 1/2 hour of lead time)
T       M       1300    230
T       M       0930    230
T       T       1300    230
T       T       1800    230
```

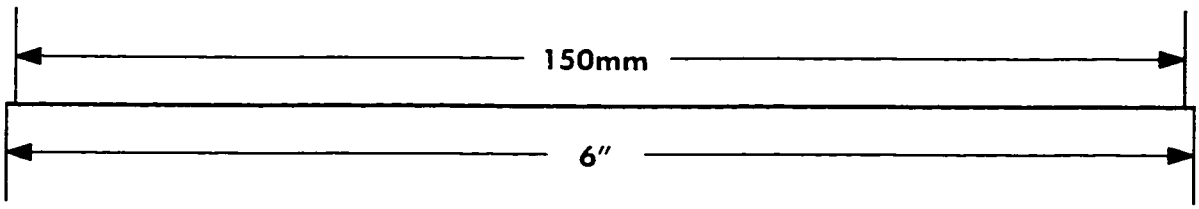
```
T       T       0930    230
T       W       1300    230
T       W       0930    230
T       R       1300    230
T       R       0930    230
T       F       1300    230
T       F       0930    230
C
```

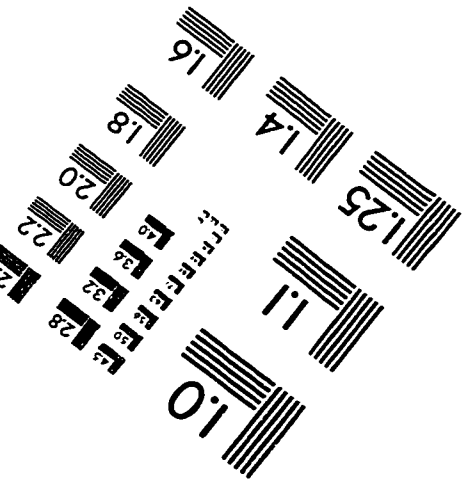
IMAGE EVALUATION TEST TARGET (QA-3)

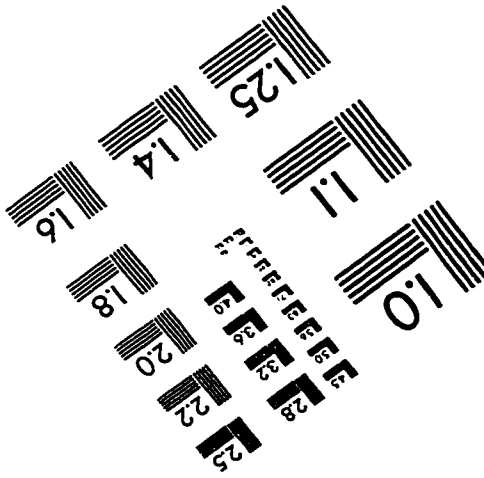












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989