Undergraduate Thesis
Bachelor of Computer Systems Engineering

Lecture Timetabling Using Genetic Algorithms

Author Leon Bambrick
Supervisor Dr B Lovell

Department of Electrical and Computer Engineering

The University of Queensland

11 Rhuddlan Street

Carindale 4152

Ph (07) 3398 7864

October 20, 1997

The Dean

School of Engineering

University of Queensland

St Lucia, Q 4072

RE: Submission of thesis entitled “Lecture Timetabling Using Genetic Algorithms”.

Dear Professor Simmons,

In accordance with the requirements of the degree of Bachelor of Engineering (Pass) in the division of Computer Systems Engineering I present the following thesis entitled “Lecture Timetabling Using Genetic Algorithms". This work was performed under the supervision of Dr B Lovell.

I declare that the work submitted in this thesis is my own, except as acknowledged, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

LEON BAMBRICK

328 828 944

Abstract.

This paper details the implementation of a computer program which employs Genetic Algorithms (GAs) in the quest for an optimal lecture timetable generator. GA theory is covered with emphasis on less fully encoded systems employing non-genetic operators. The field of Automated Timetabling is also explored. A timetable is explained as, essentially, a schedule with constraints placed upon it. The program, written in C, incorporates a repair strategy for faster evolution. In a simplified university timetable problem it consistently evolves constraint violation free timetables. The effects of altered mutation rate and population size are tested. It is seen that the GA could be improved by the further incorporation of repair strategies, and is readily scalable to the complete timetabling problem. Appendices include the entire source code.

Acknowledgments

I would like to acknowledge the contribution of Dr Brian Lovell in firstly suggesting the topic and secondly supervising this thesis through its various phases.

Also I wish to thank Ms Sharon Hennessey for her invaluable asistance and endless support.

“Everyone has a story to tell about a bad timetable.”

Burke and Ross (1995), pg 9.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

This paper explains an example usage of Genetic Algorithms (GAs) for finding optimal solutions to the problem of Lecture Timetabling at a large university. There are two objectives in this. First, to provide a detailed introduction to the topic of Genetic Algorithms- their history, their method and their variations. The second objective is to apply them to the problem of Automated Lecture Timetabling.

There are a number of papers available which discuss the matter of Automated Timetabling in detail. A fair proportion of these employ GA style methods to provide solutions that are actually in place in several institutions. A collection of these can be found in Burke and Ross (1996).

The current timetabling method used at the University of Queensland (UQ) is considered adequate. This thesis does not in any way represent a proposed replacement of that system.

Chapter 2: Background

Theory of Genetic Algorithms

The Origins of Artificial Species

John Holland's book "Adaptation in natural and artificial systems" as well as De Jong's "Adaptation of the behavior of a class of genetic adaptive systems," both published in 1975, are seen as the foundation of Genetic Algorithms (GAs) (Davis, 1991).

Holland's original schema was a method of classifying objects, then selectively "breeding" those objects with each other to produce new objects to be classified (Buckles and Petry, 1994). Created for the direct purpose of modelling Darwinian natural selection, the programs followed a simple pattern of the birth, mating and death of life forms. A top level description of this process is given in FIGURE 1 (Gen and Cheng, 1997; Buckles and Petry, 1992; Davis, 1991; Shaffer, 1996; NovaGenetica, 1997; Heitkoetter, 1993; Hoener, 1996).

```
Create a population of creatures.
Evaluate the fitness of each creature.
While the population is not fit enough:
{
        Kill all relatively unfit creatures.
        While population size < max :
        {
                Select two population members.
                Combine their genetic material to create a new creature.
                Cause a few random mutations on the new creature.
                Evaluate the new creature and place it in the population.
        }
}.
```

FIGURE 1: Top Level description of a GA.

The creatures upon which the genetic algorithm acts are composed of a series of units of information- referred to as genes. The genes which make up each creature are known as the chromosome. Each creature has its own chromosome.

A GA, as shown in FIGURE 1 requires a process of initialising, breeding, mutating, choosing and killing. The order and method of performing each of these gives rise to many variations on Holland’s original schema.

Encoding of chromosomes

“a certain amount of art is involved in selecting a good decoding technique when a problem is being attacked” (Davis 1991, p 4)

The first place one starts when *implementing* a computer program is often in choosing data types. And that is where the first major variation between Holland’s original schema and many other types of GA arises (Buckles and Petry, 1997).

Holland encoded chromosomes as a string of binary digits. A number of properties of binary encoding work to provide simple, effective and elegant GAs. There are, however, many other ways to represent a creature’s genes, which can have their own implicit advantages (Davis, 1991).

In order to get a problem into gene form, the substance of its solution must be represented as a collection of units of information (Davis, 1991). This is true of many problems. For example, when designing a weekly budget, the amount spent on each item could be stored as a number in a column. This can be thought of as not just a list of values but a string of genes. The value in the first row might represent the amount of money to spend on rice, and the second row might be the amount of money to spend on caviar and so on. Each of these values might be converted from base 10 to base 2 to create a fixed width binary number. Hence the problem of minimising your budget while maintaining your survival is translated into a genetic representation. A collection of possible budgets could be thus encoded, producing a population of Budget creatures.

In many cases, the problems to be encoded can seem more easily represented by data types other than binary (Anderson and Simpson, 1996). For example the budget we are encoding seems more similar to a list of real numbers with two decimal places than a list of binary numbers. If the performance of the available computing equipment will allow, then genes can be represented in the form of integers, reals, arrays, or any data type declarable (Davis 1991). Encoding methods which do not use a binary representation are said to be partially decoded. It is often noted that at this stage that Evolutionary Programming techniques also use a population of creatures whose external performance is not fully encoded (Heitkoetter, 1993).

A chicken and egg problem.

In the GA dialect the encoded list of genes is labelled a genotype. While the actual thing which they encode is labelled a phenotype (Gen and Cheng, 1997). Whether a phenotype is encoded to create a genotype or a genotype is decoded to create a phenotype, is a question on which the literature varies (Heitkoetter, 1993). In any case, the actual value of each gene is termed an allele (Gen and Cheng, 1997). In nature, the genes of living creature are stored as pairs and each parent only presents one gene from each pair (Sherwood, 1993). This differs from GAs, in which genes are not stored in pairs. But, in both GAs and biological life-forms, only a fraction of a parents' genes are passed to each offspring (Davis, 1991).

Population Size.

The first step in a GA is to initialise an entire population of chromosomes. The size of this population must be chosen. Depending on the available computing techniques, different sizes are optimal. If the population size chosen is too small then there is not enough exploration of the global search space, although convergence is quicker. If the population size is too large then time will be wasted by dealing with more data than is required and convergence times will become considerably larger (Goldberg, 1989).

Evaluating a chromosome

Random populations are almost always extremely unfit (Davis, 1991). In order to determine which are fitter than others, each creature must be evaluated. In order to evaluate a creature, some knowledge must be known about the environment in which it survives. This environment is the partially encoded (or partially decoded) description of the problem (Gen and Cheng). In our budgeting example we might describe the characteristics of a good budget as a collection of rules. One rule might be “Caviar is expensive and not very nourishing- any budget which spends a lot on caviar will not rate very well.” One by one each piece of knowledge relating to the problem is converted to another rule used in evaluating a creature. There might be one or more rules used in evaluating a chromosome. Where there are a number of rules (ie, in a multi-objective problem), each rule can be given a relative importance- a weighting (Rich, 1995).

Depending on the way we structure the method of evaluating a chromosome we can either aim to generate the least costly population or the most fit; it is a question of minimising cost or maximising fitness. In the budgeting example, the heuristic concerning caviar can be represented with a cost. In optimisation problems cost is not a measure of money, but a unit of efficiency (Gen and Cheng, 1997; Davis, 1991). In this case it is simpler to say that caviar is costly, than that a lack of caviar is healthy. Of course, fitness can be seen as inversely related to cost and vice versa, so one can be easily transformed to the other (Gen and Cheng, 1997).

When discussing optimisation techniques, the range of possible solutions is often referred to as the solution space and the cost/fitness of each point in the solution space is referred to as the altitude in the landscape of the problem. To looking for the global minimum of the cost is also to look for the lowest point in the lowest valley of the cost landscape. Similarly, to look for the global maximum fitness is to look for the highest point of the highest mountain in the fitness landscape. Terminology that assumes an understanding of the concept of a cost landscape will be used throughout this paper.

With any non trivial optimisation problem it would take an unreasonably long amount of time to exhaustively search the solution space for the global minimum of cost. As such optimisation techniques are employed which utilise two techniques to hasten the search, referred to as exploitation and exploration. Exploitation is when information about the explored region of the landscape is used to direct the search. Exploration is where new, unexplored regions of the landscape are ventured into (Gen and Cheng, 1997). Finding a suitable medium between these two concepts is essential for fast optimisation (Gen and Cheng, 1997).

Initialising a Population.

There are two general techniques for initialising a population. A population of creatures (all of the genetic information about all of the creatures in the colony) can be loaded from secondary storage. This data will then provide a starting point for the directed evolution. More commonly the GA can start with a random population. This is a full sized population of creatures whose genetic make up is determined by a random process (Davis, 1991).

Methods of Selecting for Extinction or for Breeding

Once a full population of creatures is established, each with a measure of fitness (or of cost) we can find an overall fitness. If the overall fitness is not yet as high as is desired a portion of the least fit creatures in the population can be selected for extinction. This can be referred to as an elitist natural selection operator (Davis, 1991).

Alternatively, an overcrowding strategy can be employed. Early GAs used a replacement strategy which maintained a constant population by replacing two parents with their two offspring in each generation (Gen and Cheng, 1997). Soon afterward a “crowding strategy” was invented which had a single offspring replacing which ever of its parents it most resembled. This can require a gene by gene comparison of the child to each of its parents, and as such is quite expensive, computationally (Gen and Cheng, 1997).

Tournament selection is another technique for deciding which creatures to eradicate. In this scheme, two creatures are chosen and played off against each other- the winner is allowed to reproduce and/or the loser is selected for extinction (Rich, 1995). This is said to mimic behaviour exhibited by stags in large deer populations and occasionally seen amongst humans (Heitkoetter, 1993).

Exactly how many creatures are wiped out at each generation is a question of some importance. The proportion of premature termination in the population creates what is termed the selection pressure (Gen and Cheng, 1997; Gell-Mann, 1994). For example, in natural life forms events such as plagues, wars, floods, famines or ice ages represent periods in which selection pressures are quite high, in varying directions in each case.

In various GAs, the method of selecting creatures for breeding is handled in different ways. Holland’s original model uses a method where the healthiest are most likely to breed (Gen and Cheng, 1997). Other methods select any two creatures at random for breeding. Selective breeding can be used in conjunction with or in the absence of an Elitist Natural Selection Operator- in either case the GA can perform evolution (Gen and Cheng, 1997).

In highly evolved populations of creatures, the process of speciation begins. This is where the intra-mating of some groups (termed “species”) causes high fitness offspring of that species, while the mating of the species members with members of the population who are not of that species produces extremely low fitness offspring, termed “lethals.” Lethals rarely survive into the next generation (Heitkoetter, 1993).

The purpose of selective breeding is both to promote high fitness chromosomes (Gen and Cheng, 1997) and to avoid the over production of lethals (Heitkoetter, 1993).

Crossover

Once parents have been chosen, breeding itself can then take place. A new creature is produced by selecting, for each gene in the chromosome, an allele from either the mother or the father. The process of combining the genes can be performed in a number of ways. The simplest method of combination is called single point cross-over (Gen and Cheng, 1997; Buckles and Petry, 1997; Davis, 1991). This can be best demonstrated using genes encoded in binary, though the process is translatable to almost any gene representation (Davis, 1991).

A child chromosome can be produced using single point crossover, as shown in FIGURE 2. A crossover point is randomly chosen to occur somewhere in the string of genes. All genetic material from before the crossover point is taken from one parent, and all material after the crossover point is taken from the other (Davis, 1991).

The process of crossover can be performed with more than one crossover point (Gen and Cheng, 1997). Indeed, every point can be chosen for crossover if preferred (Heitkoetter, 1993). One method of crossover, often used in multi-objective systems, is unity order based crossover (Heitkoetter, 1993). In this scheme, each gene has an equal probability of coming from either parent- there may be a crossover point place after each or any gene (Heitkoetter, 1993).

```
Two parents have already been selected:

PARENT1:    101101010101001001001001110011100110101011101101
PARENT2:    010100111011010101110101001001101011001010010110

Choose a crossover point:

PARENT1:    1011010101010010    01001001110011100110101011101101
PARENT2:    0101001110110101    01110101001001101011001010010110

Perform crossover to produce a child:

CHILD:      1011010101010010    01110101001001101011001010010110

Which then becomes, a whole new chromosome:

CHILD:      101101010101001001110101001001101011001010010110
```

FIGURE 2: An Example of Crossover with Fully Encoded Genes

Mutation

After crossover is performed and before the child is released into the wild, there is a chance that it will undergo mutation. The chance of this occurring is referred to as the mutation rate. This is usually kept quite small (Davis, 1991). The purpose of mutation is to inject noise, and, in particular, new alleles, into the population. This is useful in escaping local minima as it helps explore new regions of the multi dimensional solution space (Gen and Cheng, 1997). If a mutation rate is too high it can cause well bred genes to be lost and thus decrease the exploitation of the high fitness regions of the solution space. Some systems do not use mutation operators at all (Heitkoetter, 1993). Instead, they rely on the noisy (ie, diverse) random populations created at initialisation to provide enough genes that recombination alone will yield an effective search (Heitkoetter, 1993).

Once a gene has been selected for mutation, the mutation itself can take on a number of forms (Davis, 1991). This, again, depends on the implementation of the GA. In the case of a binary string representation, simple mutation of a single gene causes that genes value to be complemented- a 1 becomes a 0 and vice versa. This is analogous to the effect of stray ultra violet light upon genes in nature (Gen and Cheng, 1997). The genetic sensitivity which allows light to cause various forms of cancer also helps life on this planet to search the solution space of the ultimate question.

In the case of non binary gene representations, more cumbersome methods of mutation are required. For integer or real number representations, a common method is to add a zero mean Gaussian number to the original value. In more complex data types a value can be randomly selected from a library of possible values. In any case, all that is required is that the method of mutation is general enough that it can cause the appearance of any possible allele within the population (Davis, 1991). It seems that in more highly decoded genetic representations, mutation becomes increasingly complicated.

Inversion.

In Holland's founding work on GAs he made mention of another operator, besides selection, breeding, crossover and mutation which takes place in biological reproduction. This is known as the inversion operator (Davis, 1991). An inversion is where a portion of a chromosomes detaches from the rest of the chromosome, then changes direction and recombines with the chromosome. This is demonstrated in FIGURE 3.

Chromosome before Inversion:	001001011010100101011010010100101010001010		
During Inversion:	00100101101010	01010110100101	00101010001010
One portion inverts: (order is reversed)		01010110100101 becomes 10100101101010	
Recombination:	00100101101010	10100101101010	00101010001010
After Inversion:	001001011010101010010110101000101010001010		

FIGURE 3: The Inversion Operator

The process of inversion is decidedly more complex to implement than the other operators involved in genetic algorithms. Because of this, and because GAs can perform evolution without inversion, it is generally not used (Davis 1991). However, it is suspected that in the highly complex GA problems of the future, inversion might play a vital role (Davis, 1991). Gen and Cheng (1997) mention the use of an inversion technique of mutation (amongst others) in a GA solution to a travelling salesperson problem.

Lamarckian Operators.

In order to optimise performance of a GA, many applications also apply other techniques to the chromosomes. For example, after a chromosome has been bred, it might be possible to apply local hill climbing techniques or greedy algorithms to improve the chromosomes fitness, prior to evaluation. This is referred to as Lamarckian Evolution in Gen and Cheng (1997), as an analogy to Lamarck's theory of adaptation in which lessons learned from experiences occurring in the life of an organism can be passed on to its offspring (Gen and Cheng, 1997).

Memetic Algorithms.

Memetic Algorithms (MAs) are often explained as follows. In GAs, the unit of information is a gene, and this is generally passed on, intact, from one creature to the next. In MAs, the unit of information is a "meme," and each creature interprets/alters the meme when it is first received (Burke, Newall and Weare, 1995). This is based on an analogy with the way that information passed from person to person is interpreted and considered by each person (Gen and Cheng, 1997). This reduces the overall solution space, thereby enabling a sufficient search in less generations (Gen and Cheng, 1997). Lamarckian operators applied in conjunction with this interpretation of the meme can effectively compress a large search space into a smaller once by only considering local optima. This can decrease the number of generations required to optimise the solution, but it might also increase the time taken to perform each step (Burke, Newall and Weare, 1995).

Repair Strategies.

Some representations of genes can cause offspring that are outside the search space (Gen and Cheng). Consider a system which breeds lecture timetables. Two timetables, chosen as parents, might each have only one instance of each class. Yet, if these parents are not identical, then a child produced by their crossover might have multiple bookings of some classes. Such a timetable might be outside the search space of the problem- in which case a repair strategy could be used to remap the chromosome to within the search space. For this example, the repair strategy would alter the child's genes to ensure that exactly one booking of each class is made in a week. Another scheme involves immediately rejecting any chromosome which is outside the search space of the problem. Alternatively, there could be a large penalty cost for timetables with multiple bookings. This can be termed a penalising strategy and it requires the searching of a much larger search space. Gen and Cheng (1997) indicate that repair strategies have been shown to converge far more quickly than penalty strategies and rejection strategies.

The final stage in the genesis of a creature is to determine the creature's fitness. This process is performed in the same way as it was during initialisation (Davis, 1991).

Each iteration of the process is termed a generation. The process can terminate when it has either run for a preset number of generations (Goldberg, 1989), or it has produced a sufficiently fit population or, perhaps, at least one sufficiently fit individual creature (Heitkoetter, 1993).

Optimising Genetic Algorithm Performance

"the problem of tuning the primary algorithm presents a secondary, or metalevel, optimisation problem." (Grefenstette 1986, p 5)

It can be seen that in any implementation there are a number of variables the value of which will change the speed and effectiveness of the evolutionary process. Variables include mutation rate, selection pressure, number of crossovers, constraint weightings and so on. In more complex implementations there are a greater number of these variables. For each of these variables there is a range of values which provide a working GA. But various combinations of values, at different times, will provide better performance (Grefenstette, 1986). There are many techniques which can be used for determining optimum values for these variables (Ravise, Sebag and Schoenauer, 1995). GAs themselves are of one of these methods (Heitkoetter, 1993). It is theoretically possible to have GAs driven by GAs, ad infinitum.

Applications of Genetic Algorithms

Unlike most methods of combinatorial optimisation, GAs did not initially have an underlying mathematical model. As such, they spent some time demonstrating themselves on a number of famous mathematical problems (such as the travelling salesperson problem and the k-armed bandit problem) before tackling more practical issues (Davis, 1991).

By 1989 when David E Goldberg released the seminal “Genetic algorithms in search, optimisation and machine learning”, the field had begun the brightest phase of its career- that of Being Applicable to Real World Problems (Davis, 1991).

Any problem which can be phrased so as to require the minimising or maximising of some function can be addressed by GAs (Davis, 1991). In particular, where this function is dependent upon a great many variables, such that more conventional methods are out of their depth, evolutionary methods become attractive (Corne and Ross, 1995).

Particularly noteworthy applications of GA include the solving of pipe network optimisation problems (Anderson and Simpson, 1996) transportation problems (Gen and Cheng, 1997) conformational analysis of DNA (Davis, 1991) image processing and machine learning (Buckles and Petry, 1992) and, of course, scheduling problems (Burke and Ross 1996; Buckles and Petry, 1992).

GAs are by their very nature, easily translated to parallel systems (Davis, 1991). Each creature is to some part separate from each other creature and related, to some degree. At the moment of breeding and death, there must be some interaction between one creature and the colony (or some portion of the colony). Tournament selection is a method of choosing for extinction (or for selecting for breeding) which is most effectively executed on a parallel system. In this case, it is not necessary for any one machine to know the average fitness of the entire population, only for the machines possessing the combatants to briefly communicate. The application of GAs to parallel architectures has seen a large improvement in their performance, and has created a large amount of interest (Davis, 1991; Buckles and Petry, 1997). This appears to be the major direction in which GA is heading.

GAs are advancing by containing less of a close metaphor with natural evolution- instead conforming only to that essence of evolution which allows it to work. For example, data structures are replacing binary numbers as the most common form of representing genetic material. In modern GAs, chromosomes are rarely fully encoded (Davis, 1991).

Automated Timetabling

UQ St Lucia campus has over ten thousand students, partaking in approximately three thousand classes per semester (University of Queensland, 1997). The production of suitable timetables each semester represents a large scale optimisation problem.

Timetables at UQ are currently devised by a software application entitled Tiara, which was written by a company named Opcom. Opcom is also responsible for the TransInfo public transport information system, and a number of other large scale optimisation problems in Queensland (Opcom, 1996). The clash resolution methods which they use do not employ Genetic Algorithms.

Automated methods used to solve timetabling include Tabu Search, Simulated Annealing, Evolutionary Algorithms and Artificial Intelligence (D. de Werra, 1995). There are a number of papers within Burke and Ross (1996) that deal specifically with GA methods of automated timetabling. In Corne and Ross (1996) it is noted that only on particularly complex or resource starved timetabling problems do Evolutionary Algorithms (including GAs) begin to outperform methods such as hill-climbing.

Professional software currently available for Automated Timetabling lacks the generality required by different institutions (Burke and Ross, 1996). This can mean that code needs adjustment or lengthy training and installation programs before it can be implemented at an institution which it was not intentionally written for.

Soft and Hard Constraints.

A timetable is essentially a schedule which must suit a number of constraints. Constraints are almost universally employed by people dealing with timetabling problems (Burke and Ross, 1995). Constraints, in turn, are almost universally broken into two categories: soft and hard constraints (Burke and Ross, 1995).

Hard constraints are constraints, of which, in any working timetable, there will be no breaches. For example, a lecturer cannot be in two places at once (Erben and Keppler, 1995; Rich 1995). An extensive list of hard constraints is provided as FIGURE 4.

Soft constraints are constraints which may be broken, but of which breaches must be minimised. For example, classes should be booked close to the home department of that class (Erben and Keppler, 1995). Soft constraints mentioned in various papers show drastic differences. Also their order of importance appears to be a source of contention. Examples of some common soft constraints are provided in FIGURE 5.

In addition to constraints, there are a number of exceptions which must be taken into consideration when constructing an Automated Timetabling system. A number of these are listed in FIGURE 6.

- Classrooms must not be double booked.
- Every class must be scheduled exactly once.
- Classes of students must not have two bookings simultaneously.
- A classroom must be large enough to hold each class booked to it.
- Lecturers must not be double booked.
- A lecturer must not be booked when he/she is unavailable. For example, a lecturer might have prior commitments.
- Some classes require particular rooms. For example, experiments might be held in particular laboratories.
- Some classes require classrooms to have particular equipment. For example audio visual equipment.
- Some classes need to be held consecutively. Consider a six hour-long practical experiment.

FIGURE 4: An Extensive List of Hard Constraints.

- Some lecturers do not wish to have classes assigned consecutively in time.
- There are preferred hours in which a lecturer’s classes might be scheduled.
- The distance a lecturer walks should be minimised
- Most students and some lecturers do not wish to have empty periods in their timetables.
- Classes should be distributed evenly over the week.
- Classrooms should be booked close to the home department of that class.
- Classrooms should not be booked which are much larger than the size of the class.

FIGURE 5: A List of Soft Constraints.

A timetable might cover more than one campus
Timetable requirements might vary from one week to the next.
More than one member of staff might need to be assigned to a particular class.
A classroom is not needed for a field trip.
Interdisciplinary subjects might be studied.

FIGURE 6: A Short List of Exceptions

Other Applications of a Lecture Timetabling GA

Exam Timetabling is one of the most closely related disciplines to Lecture Timetabling, and it is also approachable using GA methods (Burke and Ross, 1996). Essentially the same structure as used in a GA Lecture Timetabling system can be applied to the Exam Timetabling problem (Burke, Newall and Weare, 1995). The rules used for evaluating each proposed exam timetable must be rewritten accordingly. Minimising of an individual's stress is particularly important in exam timetabling, so rules such as "No student should sit two exams in the one day" are given high priority (Ergul, 1995). In Exam Timetabling the exact enrolment is known (unlike in Lecturer Timetabling where it is based on predicted enrolment). Considerations must be therefore be made not just on the basis of each class, but on an individual student basis (Ergul, 1995). This greatly increases the size of a chromosome, and creates a considerable difference between the performance parameters of Lecture and Exam Timetabling GAs.

Wherever there is a maximum stress, minimum resource (Rich, 1995) multiple objective timetabling problem then a GA similar to a Lecture Timetabling system might be applicable (Corne and Ross, 1995). Consider the problems of scheduling of large transport systems, rostering in large corporations, multi-level job-scheduling in operating systems the timing constraints of intricate digital circuits and so forth (Davis, 1991).

CHAPTER 3: METHOD

The Simplified Lecture Timetabling Problem.

The Automated Timetabling system devised in this thesis deals with a simplified version of the University Timetabling problem. The rationale for simplification, however, is that the program is suitably complex that it could be scaled to the full problem without any need to modify its architecture. This is done in two ways. Firstly, only a fraction of the hard constraints are placed on the timetables which it produces. Those considered are shown in FIGURE 7. Soft constraints are not incorporated into this GA. The theory behind adding all of the constraints (including the soft constraints) is included later. Secondly, the size of the fictional university with which we are dealing is only a fraction of the UQ timetable.

Classrooms must not be double booked.
Every class must be scheduled exactly once.
Classes of students must not have two bookings simultaneously.
A classroom must be large enough to hold each class booked to it.
Lecturers must not be double booked.
A lecturer must not be booked when he/she is unavailable. For example, a lecturer might have prior commitments.

FIGURE 7: Hard Constraints Used in The Simplified Timetabling Problem.

Implementing a Timetable as a Genetic Algorithm.

In this thesis a C program was developed which employed GA methods to perform Automated TimeTabling. Consequently, the program was entitled “gaatt.c” The complete source code of this program is included as Appendix B.

The GA operates upon a population of timetables which are maintained in memory. Each timetable is evaluated by testing the number of times it breaches each constraint. Thus timetables are evolved with a minimum number of constraint violations. A top level description of “gaatt.c” is provided as FIGURE 8. It can be seen that this structure is similar to the pseudo code given in FIGURE 1, with the major difference being the incorporation of a repair strategy. Explanation of each of the components of the program are given in the remainder of this section.

```
Load all constraint data from a constraint file.
While the population size is less than the maximum:
{
        Create a new timetable with no classes booked to it.
        Repair the new timetable by using the constraint data.
        Evaluate the cost of the new timetable by using the constraint data.
        Enter the new timetable into the population.
}
While the cost of the best timetable is greater than zero:
{
        Discard a portion of costly timetables.
        Repeat until the population size is maximum:
        {
                Breed a new timetable.
                Mutate the new timetable.
                Repair the new timetable by using the constraint data.
                Evaluate the cost of the new timetable by using the constraint data.
                Enter the new timetable into the population.
        }
}.
```

FIGURE 8: Top Level Description of “gaatt.c”

Constraint Data

In order to test for each of the types of hard constraint it is necessary to store sufficient detail about the university. This means that information concerning all lecturers, classrooms and classes must be maintained. The way in which each of these data types are implemented will now be given in detail.

A lecturer is a structured data type with one field- an availability timetable. An availability timetable is an array which indicates (using 1s and 0s) whether the lecturer is available or unavailable to lecture during each hour in the week. In this way prior commitments can be taken into consideration.

A class is a structured type with three fields. Each class has a certain size (predicted size), a lecturer number and a number indicating the code number of the group of related classes to which it belongs. For example, core first year engineering subjects might form one set of related classes, and would each have the same number in their related class field. Each class can only be listed as belonging to one set of related classes. This is a simplification which ignores more complex relations between classes, for example, where classes are studied by more than one faculty.

In this simplified GA a classroom has only one field, the capacity of the room.

The set of all lecturers is stored as an array. Similarly, there are arrays of classes and of classrooms. Each of these elements is identified uniquely by its position in the relevant array.

The set of all information concerning lecturers, classes and classrooms is termed the constraint data. Each time "gaatt.c" executes it loads all constraint data from a text file with a ".ctr" extension. Each different ".ctr" file describes a different university problem.

It should be noted that information considering each individual student is not used in this Lecture Timetabling System. This is because timetables must presumably be produced at a time of year when enrolments are not yet finalised. In fact students are to some extent expected to use class timetables of the semester to aid in the decision of which subjects they enrol in (University of Queensland, 1997). As such individual students cannot be considered. Class sizes, for example, would be determined by an expected number of students.

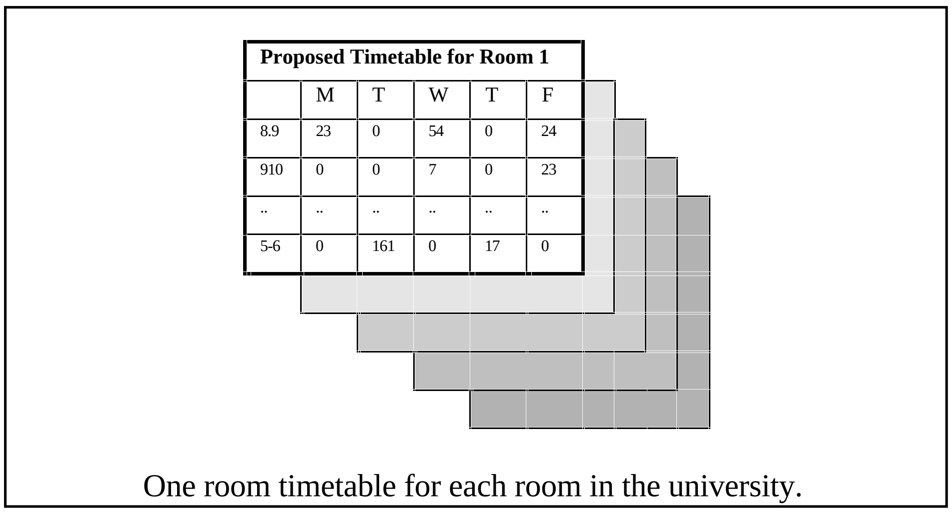
Timetables.

The timetable for a single room is a two dimensional array as shown in FIGURE 9. The timetable for an entire university is therefore a collection of room timetables- one for every room in the university- as shown in FIGURE 10. Times at which there is no class booked hold a NULL booking, which has a value of zero.

Proposed timetable for Room 1

Time	Mon	Tue	Wed	Thu	Fri
8-9	23	0	54	0	34
9-10	12	0	7	23	9
...	...	...	...	...	...
5-6	0	161	0	17	0

FIGURE 9: Example of a single room timetable.

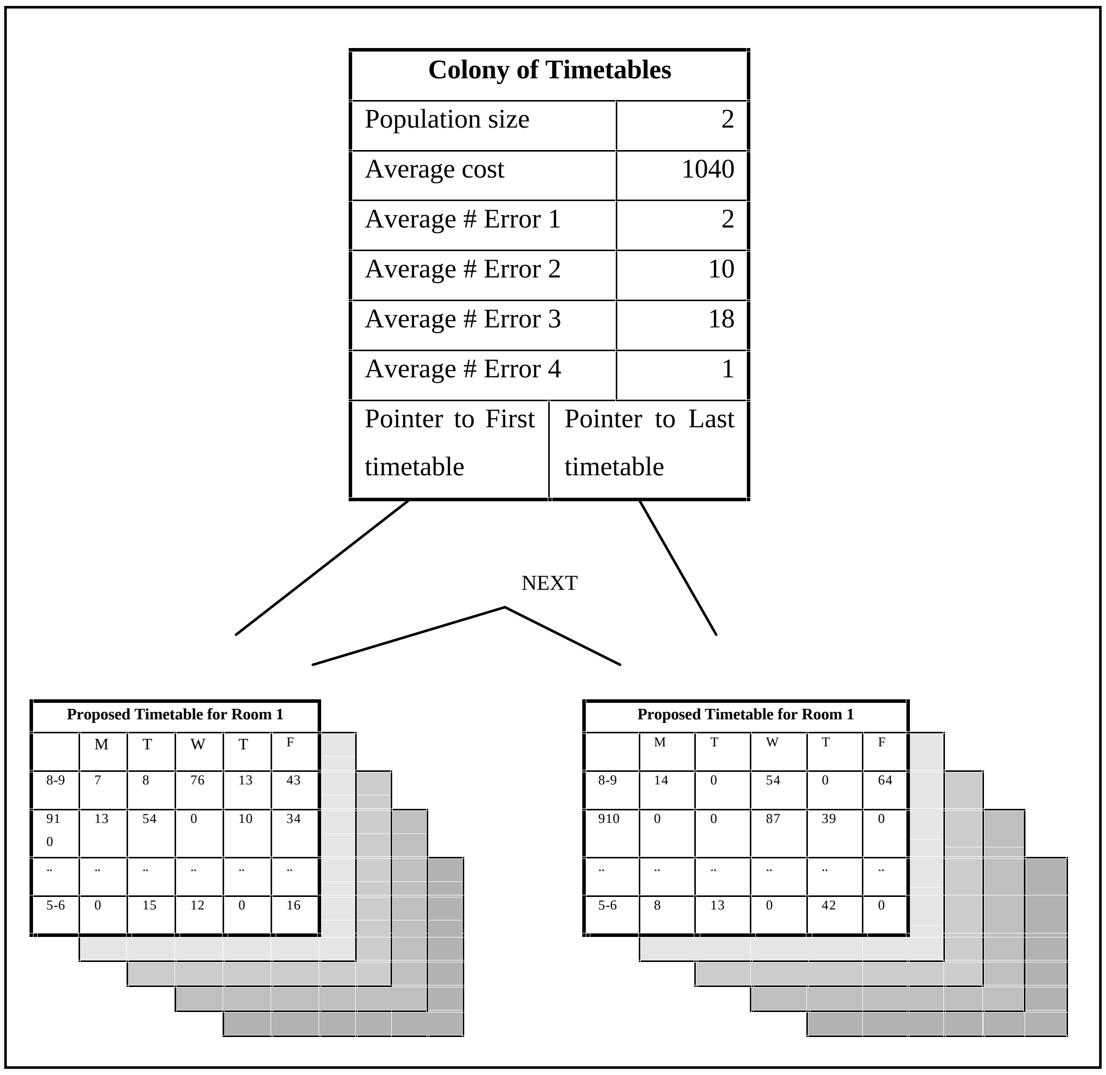


Proposed Timetable for Room 1

	M	T	W	T	F
8.9	23	0	54	0	24
910	0	0	7	0	23
..	..	..	..	..	..
5-6	0	161	0	17	0

One room timetable for each room in the university.

FIGURE 10: An Entire University Timetable.



Colony of Timetables	
Population size	2
Average cost	1040
Average # Error 1	2
Average # Error 2	10
Average # Error 3	18
Average # Error 4	1
Pointer to First timetable	Pointer to Last timetable

Proposed Timetable for Room 1

	M	T	W	T	F
8-9	7	8	76	13	43
91 0	13	54	0	10	34
..	..	..	..	..	..
5-6	0	15	12	0	16

Proposed Timetable for Room 1

	M	T	W	T	F
8-9	14	0	54	0	64
910	0	0	87	39	0
..	..	..	..	..	..
5-6	8	13	0	42	0

FIGURE 11: A Colony Containing Two Timetables

A university timetable stores information about what classes are booked in each room, at any hour of the day, on any day of the week. Each of these bookings (or NULL bookings) is one gene. A timetable also has fields which describe (decode) some aspect of this genetic information. A timetable has a field which stores its cost. It also has fields which store the number of breaches of each type of hard constraint.

A population (or colony) is a collection of timetables. A population is itself a structured type with a number of fields. It contains a pointer to the least costly timetable in the population, (which has, in turn, a pointer to the next least costly). There is also a pointer to the most costly timetable in the population, as well as a field storing the average cost, and the average number of violations of each type of hard constraint. A field is also included which records the number of timetables in the population. FIGURE 11 shows the way in which a population is comprised of a linked list of timetables.

A colony of creatures is therefore a singly-linked list of structured types (creatures) containing timetable data (genes) in a three dimensional array. The timetables are kept in order from least costly to most costly.

This method of gene representation means that it is not possible to have two classes booked to the same room at the same time. As such there is one less hard constraint to be considered when evaluating timetables.

Repair Strategy

A repair strategy is used which ensures that all classes appear exactly once. For robustness, this is done in two stages. Firstly, any classes which appear more than once are (non deterministically) altered such that they appear only once, as outlined in FIGURE 12. Secondly, any classes which did not appear at all are booked to a spare space (regardless of room size, etc) as outlined in FIGURE 13.

```
For each class:
    set the Count to 0.
    for each time:
        for each room:
            If the current class is booked at this location:
                Add 1 to the count.
                Add the location of this class to a linked list.
    If the class occurred more than once then:
        keep doing the following until there is only one booking left:
            randomly choose one of the bookings.
            turn it into a NULL booking.
    Free the linked list.
```

FIGURE 12: Pseudo Code for the First Stage of the Repair Strategy

```
For each class:
    For each time:
        Look in each room until either the class is seen or you get to the end.
        If you got to the end without finding the class then randomly find a
            NULL booking and book that class to it.
```

FIGURE 13: Pseudo Code for the Second Stage of The Repair Strategy

If this repair strategy is applied to an empty timetable the result is a timetable with each class booked to a random time and place. As such, the repair strategy is also used for initialising a random population.

The use of the repair strategy ensures that each class is booked exactly once. Hence, the number of hard constraints which must be considered when timetables are being evaluated is further reduced.

It has so far been shown that the hard constraints “classrooms must not be double booked” and “every class must be scheduled exactly once” have been satisfied by non genetic means.

Evaluation of a Timetable

The remaining for types of hard constraints are used in the evaluation of timetables. They are each considered in turn, as shown in the pseudo code of FIGURE 14. This method could be extended to any amount of hard constraints. Soft constraints violations are not calculated in “gaatt.c”. If they were, then they would be evaluated in precisely the same manner as hard constraints. This concept is explored thoroughly in the discussion section.

```
For the timetable being evaluated:
    Initialise the cost field to zero.
    For each of the hard constraints:
        Record how many times that constraint is violated.
        Add the count (multiplied by the weighting for that particular
constraint)                 to the timetable’s cost field.
```

FIGURE 14: Pseudo Code for Using Multiple Constraints to Evaluate a Timetable

Each type of hard constraint will now be considered in turn. Through out the remainder of this paper these four constraints are always considered in the same order. This is purely for the sake of consistency.

Related Class Clash Errors

For each timetable it must be determined how many times related classes are booked simultaneously. The method for doing this is given in FIGURE 15. An error of this sort implies that an entire class load of students is expected to be in two places at once.

```
For each group of related classes we must check:
    For each time:
        For each room:
            Does the class booked at this time belong to the current group
                of related classes? If so then add “1” to the Count.
        If (Count>1) then
            Record (Count-1) more “Related class clash” errors.
```

FIGURE 15: Pseudo Code for Assessing the Number of Related Class Clash Errors

Room Too Small Errors.

The number of Room Too Small Errors must also be evaluated. This calculates how many classes in a timetable are booked to rooms which are too small to accommodate them. A pseudo code description of the search for Room Too Small errors is given in FIGURE 16.

```
For each room we must check:
    For each time:
        For each class, given that the room is THIS big
            Is the size of the class bigger than THIS? If so then add “1” to
                the Count.
Record (Count) more “Room too small” errors.
```

FIGURE 16: Pseudo Code for Assessing the Number of Room Too Small Errors

Lecturer Double Booked Errors.

A count is performed to see how often lecturers are expected to be in two (or more) lectures at once. Pseudo code for this is given as FIGURE 17.

```
For each lecturer we must check:
    For each time:
        For each class:
            Is this class taught by the current lecturer? If so then add “1” to
                the Count.
    If (Count>1) then
        Record (Count-1) more “Lecturer double booked” errors.
```

FIGURE 17: Pseudo Code for Assessing the Number of Lecturer Double Booked Errors

Lecturer Unavailable Errors.

As well as lecturers having more than one lecture at a time, there is the possibility that they will have lectures at times when they have prior commitments. To penalise such cases a test is performed which counts all instances when they are booked to give a lecture at a time when they have indicated that they have prior commitments. Pseudo code for this is provided in FIGURE 18.

```
For each lecturer we must check:
    For each time, given that at that time the Lecturer's "Availability chart" shows
        either a 1 (available) or 0 (unavailable):
        For each room:
            Is the lecturer of that class the current lecturer?
            If so then if the lecturer is unavailable add "1" to the Count.
Record (Count) more "Lecturer Unavailable" errors.
```

FIGURE 18: Pseudo Code for Assessing the Number of Lecturer Unavailable Errors

Selecting Timetables for Extinction

An elitist natural selection operator is used. Because the timetables are kept in an ordered linked list this is particularly easy to implement. In each generation a fixed portion of the timetables are eradicated. For the purposes of this thesis the portion was maintained at 50%. Pseudo code of this elitist selection process is provided as FIGURE 19.

```
Work out how many timetables are going to survive.
Sort through the list until you find the last one to survive.
Assign that one as now being last in the population.
Systematically free each of the remaining timetables from the memory.
```

FIGURE 19: Pseudo Code for Selecting Timetables for Extinction

Breeding Timetables

Timetables are randomly selected from the population and used for breeding. No favouritism is given to fitter timetables. A child timetable is bred by performing unity order based crossover on the parents. This means that each parent has an equal chance of providing each gene.

Mutating Timetables

The method of mutation is given in FIGURE 20. This means that the chance of any one gene undergoing mutation is approximately twice the mutation rate divided by one thousand. A mutation rate equal to ten, for example, implies that approximately twenty in every thousand genes will be mutated.

```
There is a fixed mutation rate.
For each gene
{
    Randomly choose a number between 1 and 1000.
    If the number is less than the mutation rate then
    {
        Randomly choose a gene from the current timetable and swap it with
            the current gene.
    }.
}.
```

FIGURE 20
Pseudo Code for Method of Mutation

This method should be scalable to the complete problem. The incorporation of further constraints, and the upscaling of the problem size should not require any changes to the overall architecture.

Experimental Procedure

There were seven tests performed on “gaatt.c”. The first five tests dealt with evaluating the performance of the GA with different weightings on the constraints. Test 6 looked at the effect of the rate of mutation on the speed of evolution. Test 7 looked at the effect of the population size on the speed of evolution.

For the purposes of these tests constraint data were created at random using a Pascal program which would output valid text files for "gaatt.c" to read. The parameters of this constraint data were as follows. There were 200 classes, half of which belonged to any one of fifty related class groups. There were fifty lecturers, lecturing an average of four classes, and available for lecturing throughout seventy percent of the working week. There were fifteen rooms with sizes evenly distributed between 300 and 50 seats. The class sizes were randomly chosen, with equal probability of selection at all sizes between 0 and 250.

Tests 1 to 5: The Effect of Several Different Constraint Weightings.

In the first five tests the population size was kept at three and the mutation rate was kept at sixteen. The weightings used are as shown in Table 1.

TABLE 1: WEIGHTINGS OF EACH TYPE OF HARD CONSTRAINT VIOLATION IN TESTS 1 TO 5

	Test 1	Test 2	Test 3	Test 4	Tests 5
Related Class Clash Errors	1	0	0	0	1
Room Too Small Errors	0	1	0	0	100
Lecturer Double Booked Errors	0	0	1	0	1
Lecturer Unavailable Errors	0	0	0	1	20

Test 1: Related Class Clash Errors.

The purpose of test 1 was to see how well "gaatt.c" performed when only Related Class Clash Errors were considered. This should mean that optimisation proceeds in the direction of timetables with no Related Class Clash Errors. As such, all other types of errors were given no weighting- in essence they were left out. "gaatt.c" was executed until it terminated correctly (ie, evolved a timetable with a cost of zero) or it stagnated. The average cost of the population after each generation was recorded. This test was repeated until results were clear.

Test 2: Room Too Small Errors.

In Test 2 only Room Too Small Errors were considered. "gaatt.c" was executed until it terminated correctly (ie, evolved a timetable with a cost of zero) or it stagnated. The average cost of the population after each generation was recorded. This test was repeated until results were clear.

Test 3: Lecturer Double Booked Errors.

In Test 3 only Lecturer Double Booked Errors were considered. "gaatt.c" was executed until it terminated correctly (ie, evolved a timetable with a cost of zero) or it stagnated. The average cost of the population after each generation was recorded. This test was repeated until results were clear.

Test 4: Lecturer Unavailable Errors.

In Test 4 only Lecturer Unavailable Errors were considered. "gaatt.c" was executed until it terminated correctly (ie, evolved a timetable with a cost of zero) or it stagnated. The average cost of the population after each generation was recorded. This test was repeated until results were clear.

Test 5: Test of Four Weighted Hard Constraints.

In Test 5 all four types of hard constraint error were considered. Their weightings were: 1, 100, 1, 20 respectively. These weightings were chosen as they were thought to reflect the difficulty of removing each type of error, respectively. "gaatt.c" was executed until it terminated correctly (ie, evolved a timetable with a cost of zero) or it stagnated. The average cost of the population after each generation was recorded. This test was repeated until results were clear.

Test 6 and Test 7.

The sixth and seventh test also employed "gaatt.c". Weightings used on the hard constraints were the same as for the fifth test. That is: 1, 100, 1 and 20 for Related Class Clash Errors, Room Too Small Errors, Lecturer Double Booked Errors and Lecturer Unavailable Errors respectively.

Test 6: Mutation Rate Test.

The population size was fixed at three, and the mutation rate was varied.

With a mutation rate of 0, the GA was run until five hundred timetables had been generated and the average cost of the resultant population was recorded. This procedure was then repeated with many mutation rates between 0 and 500. Extra trials were performed where results were unclear.

Test 7: Population Size Test.

The mutation rate was fixed at sixteen and the population size was varied.

With a population of three, the GA was run until more than 500 timetables had been generated, and the average cost of the resultant population was recorded. This procedure was then repeated with every population size up to 30. Memory limitations stopped the test from going further. The test was repeated until results became clear.

Chapter 4: Results of Tests on “gaatt.c”

Tests 1-5

FIGURES 21 to 29 show the performance of “gaatt.c” in tests 1 to 5. Further information is given in Table 2, and in the following descriptions.

Test 1: Related Class Clash Errors

Test 1 was repeated 15 times. In each case “gaatt.c” successfully eradicated all instances of Related Class Clash Errors. This took an average of 55.1 generations, with a standard deviation of 24.4 generations.

The average number of Related Class Clash Errors after each generation was recorded, for each iteration of the test. These results were then averaged and are displayed in FIGURE 21.

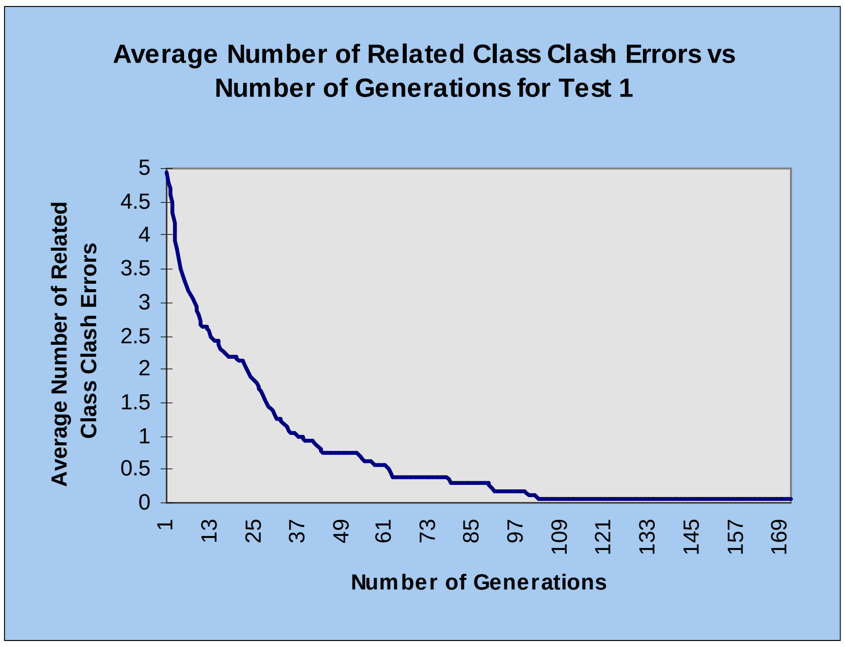


FIGURE 21: Average Number of Related Class Clash Errors vs Number of Generations for Test 1

Test 2: Room Too Small Errors

Test 2 was repeated 11 times. In each case “gaatt.c” successfully eradicated all instances of Room Too Small Errors. This took an average of 1270.6 generations, with a standard deviation of 177.9 generations.

The average number of Room Too Small Errors after each generation was recorded, for each iteration of the test. These results were then averaged and are displayed in FIGURE 22.

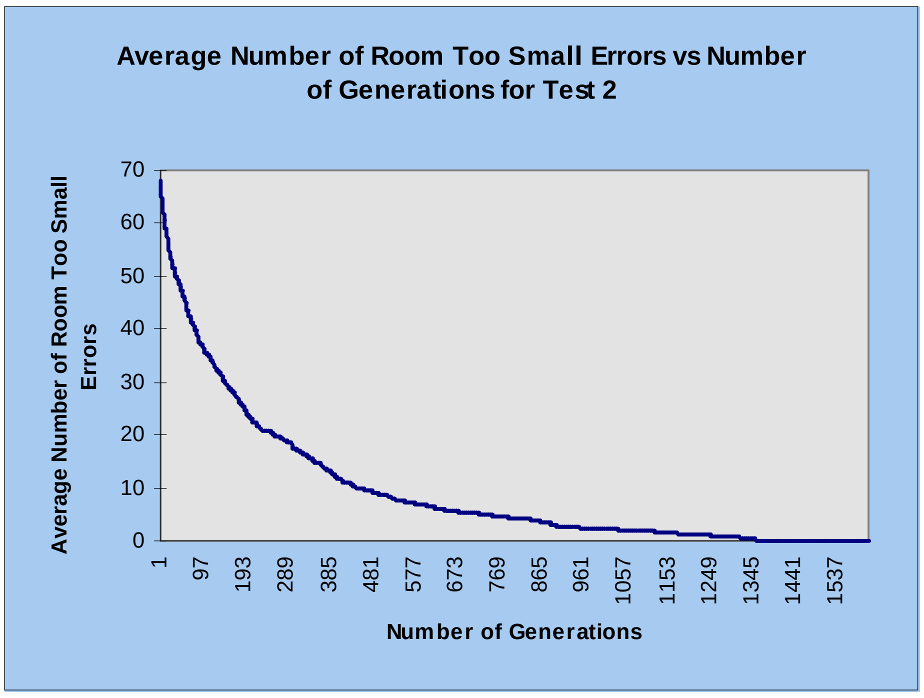


FIGURE 22: Average Number of Room Too Small Errors vs Number of Generations in Test 2

Test 3: Lecturer Double Booked Errors

Test 3 was repeated 16 times. In each case "gaatt.c" successfully eradicated all instances of Lecturer Double Booked Errors. This took an average of 59.8 generations, with a standard deviation of 39.5 generations.

The average number of Lecturer Double Booked Errors after each generation was recorded, for each iteration of the test. These results were then averaged and are displayed in FIGURE 23.

Test 4: Lecturer Unavailable Errors

Test 4 was repeated 12 times. In each case "gaatt.c" successfully eradicated all instances of Lecturer Unavailable Errors. This took an average of 293.1 generations, with a standard deviation of 73.0 generations.

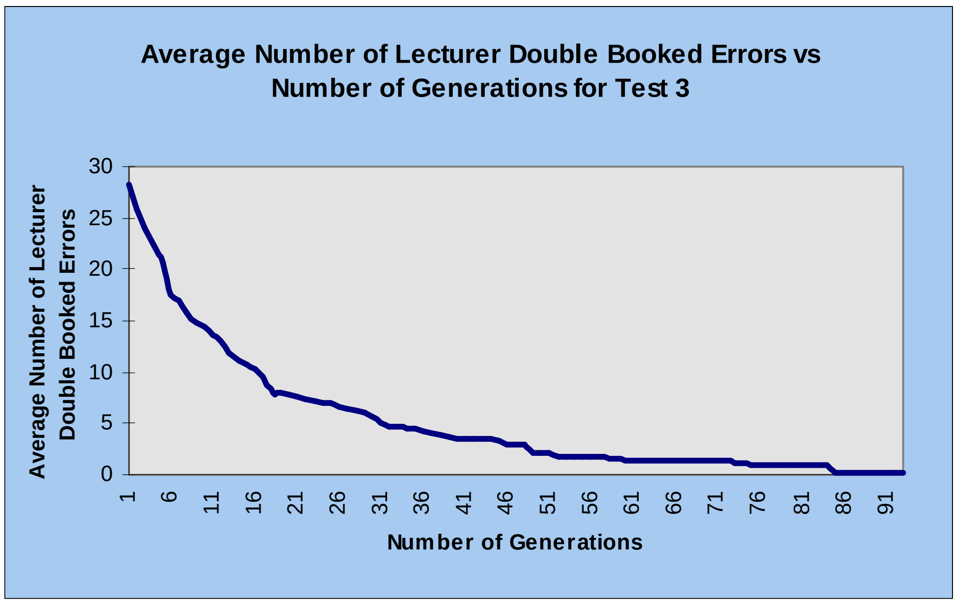


FIGURE 23: Average Number of Lecturer Double Booked Errors vs Number of Generations for Test 3

The average number of Lecturer Unavailable Errors after each generation was recorded, for each iteration of the test. These results were then averaged and are displayed in FIGURE 24.

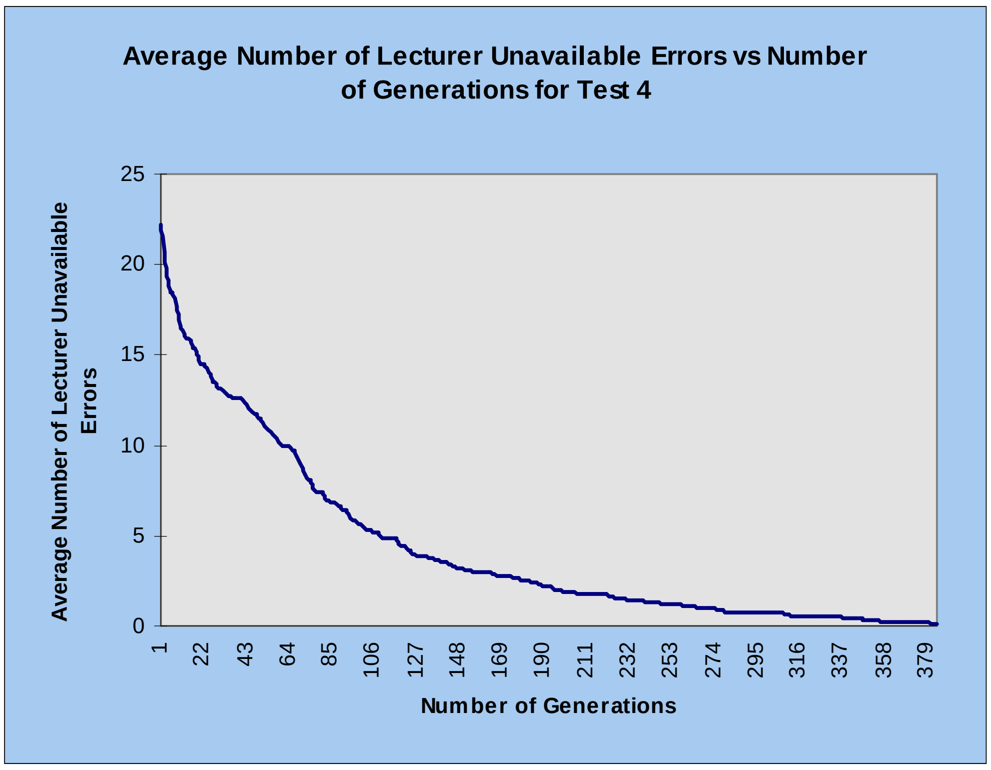


FIGURE 24: Average Number of Lecturer Unavailable Errors vs Number of Generations in Test 4

Test 5: Test of Four Weighted Hard Constraints.

Test 5 was repeated 6 times. In each case "gaatt.c" successfully eradicated all instances of Hard Constraint Violations. This took an average of 4098.2 generations, with a standard deviation of 606.5 generations,

The average cost of the population after each generation was recorded, for each iteration of the test. These results were then averaged and are displayed in FIGURE 25.

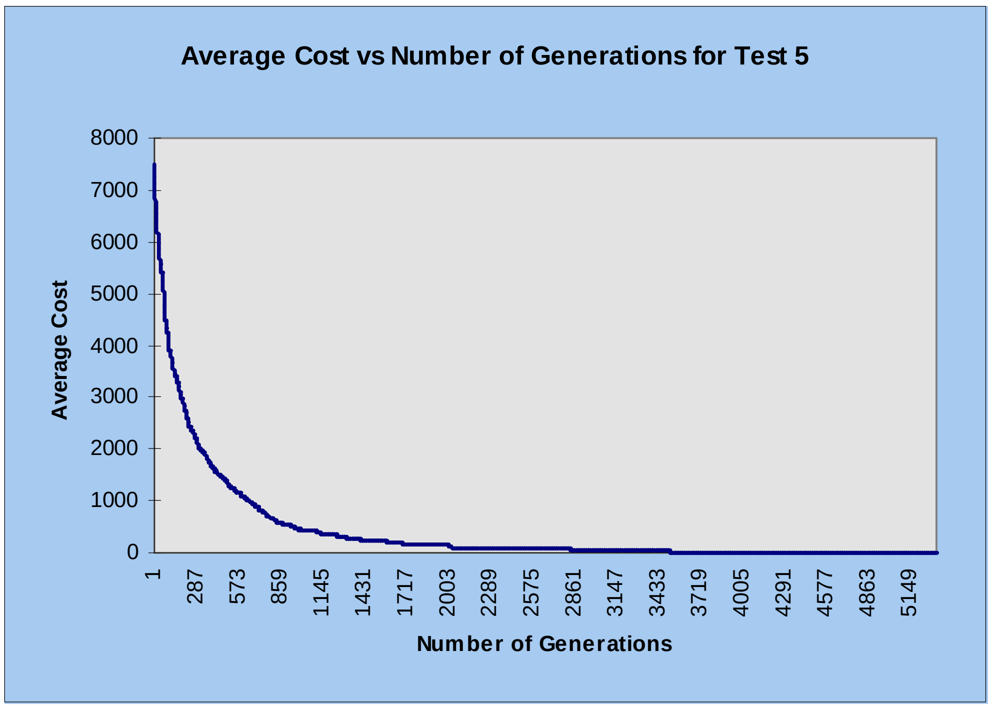


FIGURE 25:Average Cost vs Number of Generations for Test 5

Also, the average number of Related Class Clash Errors after each generation was recorded, for each iteration of the test. These results were then averaged and are displayed in FIGURE 26.

The average number of Room Too Small Errors after each generation was recorded, for each iteration of the test. These results were then averaged and are displayed in FIGURE 27.

The average number of Lecturer Double Booked Errors after each generation was recorded, for each iteration of the test. These results were then averaged and are displayed in FIGURE 28.

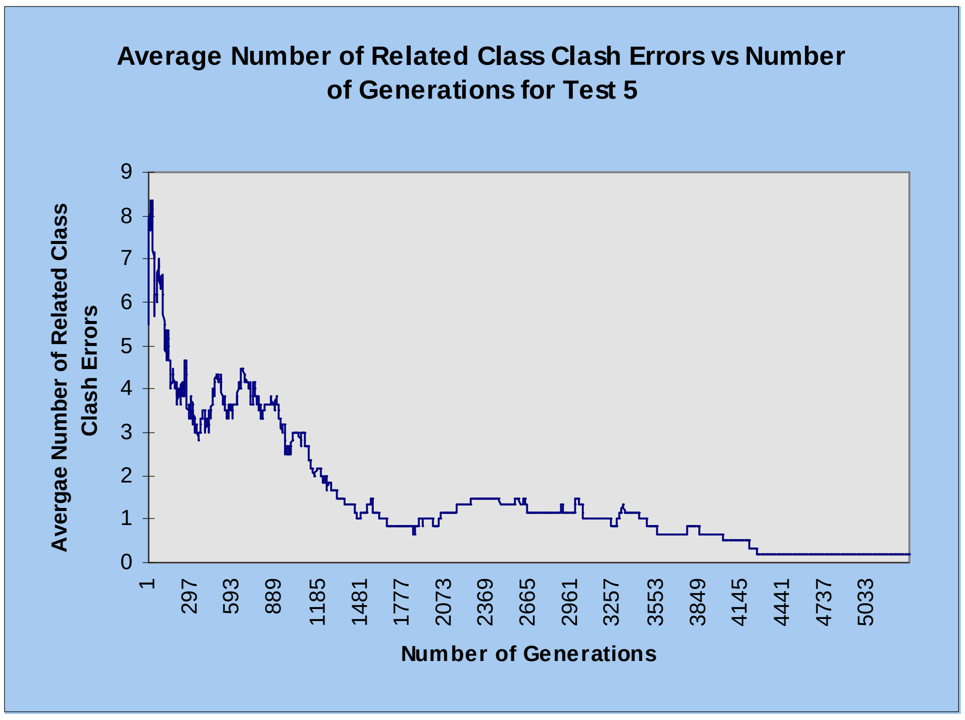


FIGURE 26: Average Number of Related Class Clash Errors vs Number of Generations for Test 5

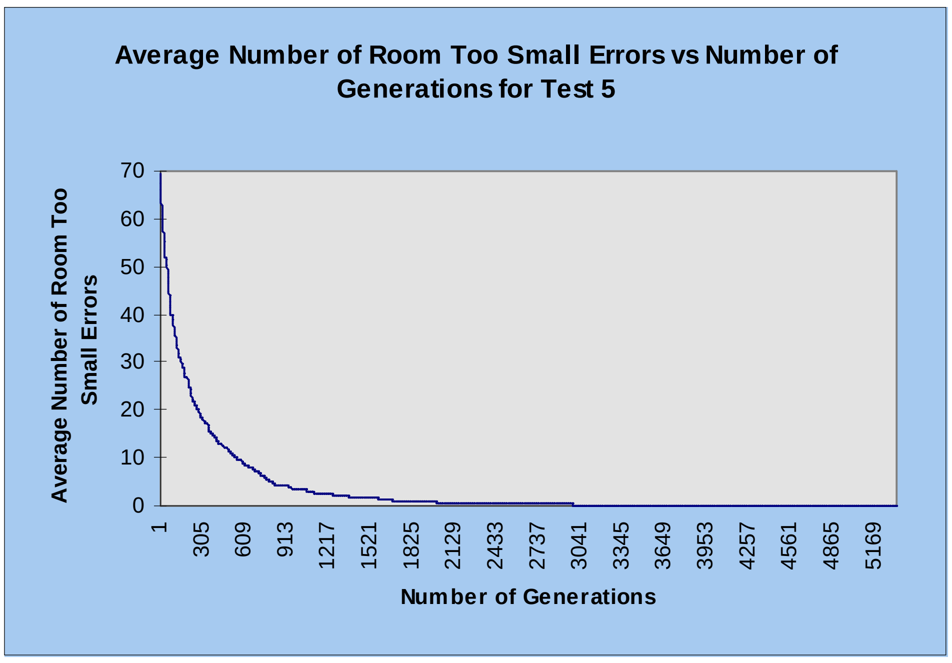


FIGURE 27: Average Number of Room Too Small Errors vs Number of Generations in Test 5

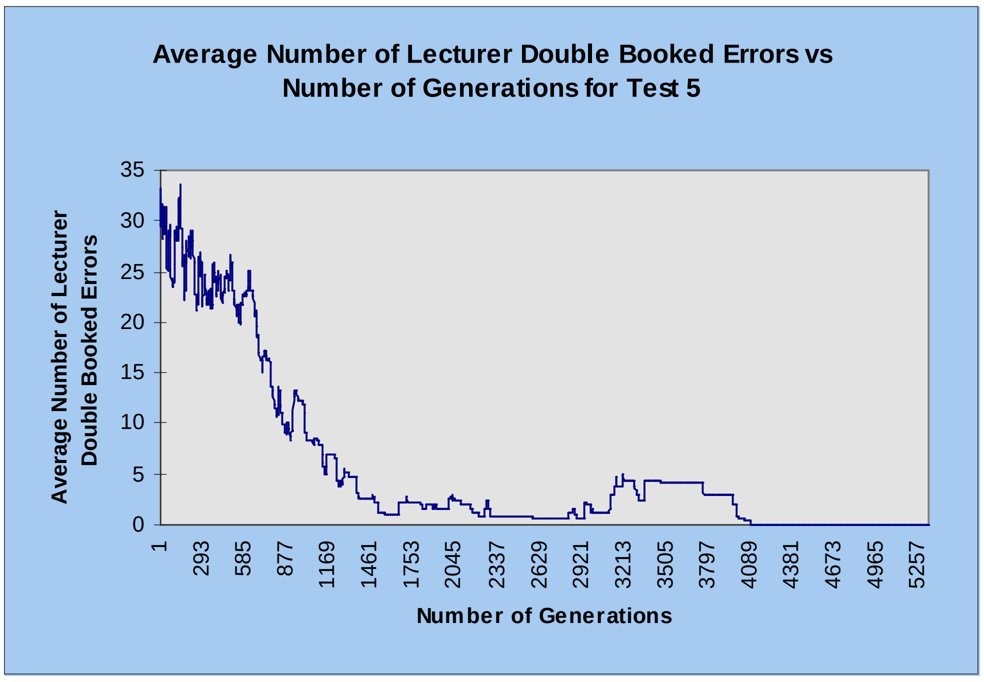


FIGURE 28: Average Number of Lecturer Double Booked Errors vs Number of Generations for Test 5

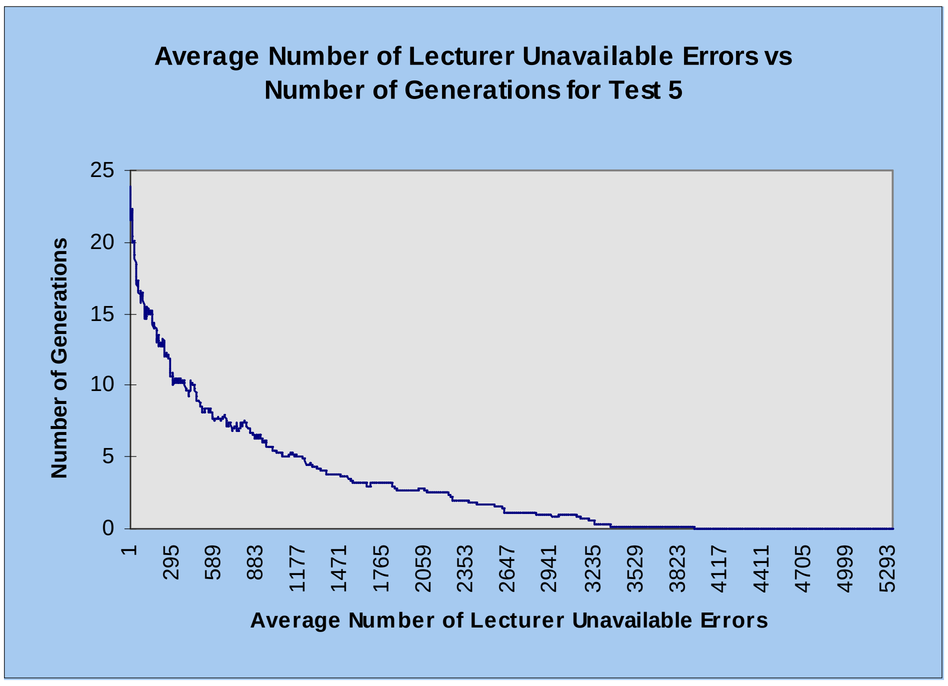


FIGURE 29: Average Number of Lecturer Unavailable Errors vs Number of Generations for Test 5

The average number of Lecturer Unavailable Errors after each generation was recorded, for each iteration of the test. These results were then averaged and are displayed in FIGURE 29.

Test 5 was also used to determine the speed at which generations were run. Tests were performed on a 100 MHz 5x86. The average time taken to evolve a cost free solution was thirteen minutes and eight seconds. It therefore took approximately 0.192 seconds to explore each timetable in the solution space. Note that this includes the time taken to progressively store the cost of each generation in a text file, so the performance of the GA itself would be moderately faster.

TABLE 2: SUMMARY OF RESULTS FROM TESTS 1 TO 5

	Test 1	Test 2	Test 3	Test 4	Test 5
Number of times test was repeated	15	11	16	12	6
Mean number of Generations required to reach cost of 0,	55.1	1270.6	59.8	293.1	4098.2
with Standard Deviation	24.4	177.9	39.5	73.0	606.5

Test 6: Mutation Rate Test.

Test 6 was performed for fifty different mutation rates ranging between zero and 500. Most of the rates tested were in the region zero to fifty, because the output from this region showed the most variation. The test was repeated four times at each mutation rate chosen. A graph of the average cost of the timetable after five hundred generations versus mutation rate is given as FIGURE 30.

Test 7: Population Size Test.

Tests 7 was repeated three times for all populations from 3 to 30. Sizes outside this range could not be implemented on the current system. A graph of the average cost of the timetable after five hundred generations versus mutation rate is given as FIGURE 31.

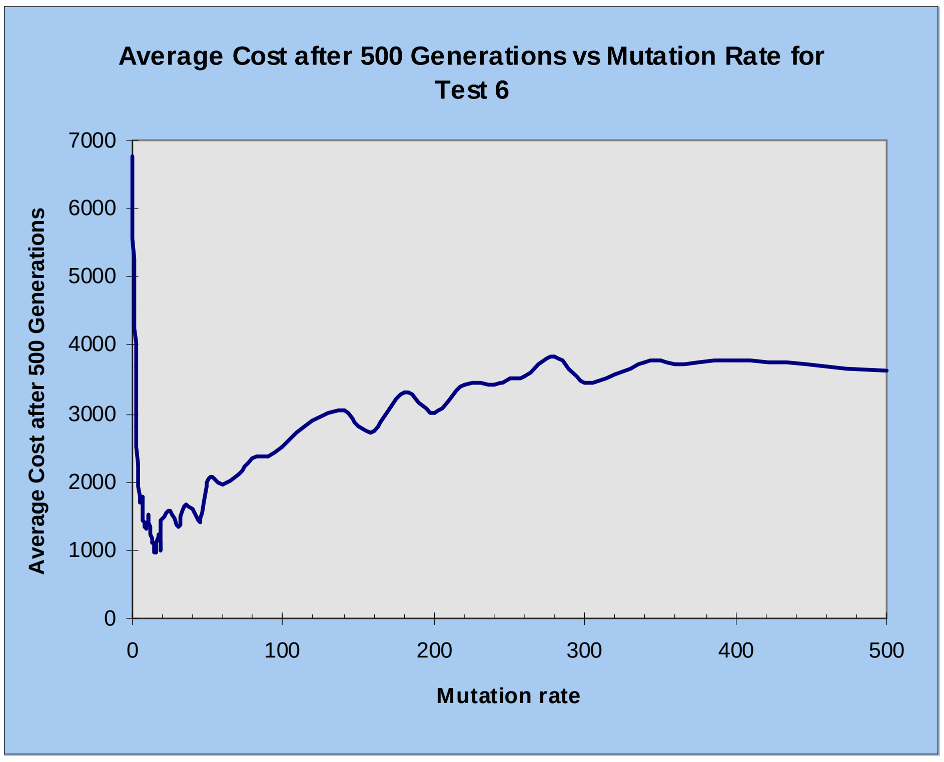


FIGURE 30: Average Cost after 500 Generations vs Mutation Rate for Test 6

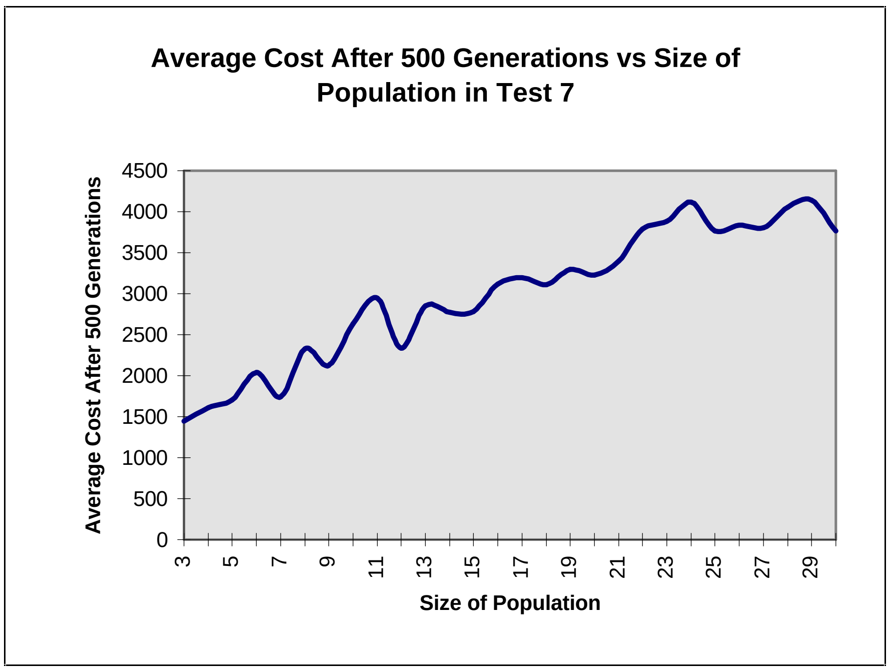


FIGURE 31: Average Cost after 500 Generations vs Size of Population for Test 7

Chapter 5: Discussion

Review of performance of “gaatt.c”

The most important question that can be asked of any Automated Timetabling system is whether or not it is capable of producing timetables completely free of hard constraint violations. The results of Test 5 showed that “gaatt.c” succeeded in this regard. In every trial the program converged on a suitable population. On average this required that 4098.2 timetables were searched. This is a pleasing result which demonstrates the power of GA techniques on Automated Timetabling Problems.

Performance of “gaatt.c” in Tests 1 to 4

Tests 1 to 4 were each concerned with the reduction of one type of hard constraint violation (or “error”). In each case, “gaatt.c” successfully removed all instances of the error in question.

As can be seen in TABLE 2 this took a differing amount of generations for each type of error. Room Too Small Errors took decidedly longer to eradicate than did each of the other error types of error. Lecturer Unavailable Errors took considerably longer than did Related Class Clash Errors and Lecturer Double Booked Errors.

One possible reason for these two types of errors taking so much longer to be eradicated relates to their cause. Both Room Too Small Errors and Lecturer Unavailable Errors are single gene errors. They are caused by single alleles. Related Class Clash Errors and Lecturer Double Booked Errors, by contrast, are caused by the destructive interaction of two or more alleles.

Another reason which must be considered for why Room Too Small Errors and Lecturer Unavailable Errors took a long time to be eradicated is that this result is specific to the constraint data. The random allocation of class sizes for the constraint data, for example, allowed unrealistic class size distributions to be created. Any possible future versions of this project might be able to use constraint data from real institutions, in order to avoid any such problems. For simplified problems, such as this, constraint data could be modelled on smaller institutions such as high schools, or single faculties of a university.

Performance of "gaatt.c" in Test 5

The combination of all four types of hard constraint error displayed an anti gestalt property, in that the number of generations to solve for all four constraints simultaneously was greater than to solve for the sum of each constraint individually. This is not an unexpected result. The interaction between the four type of error means that there are less directions in the cost landscape in which the population can evolve.

FIGURE 25 resembles a scaled version of FIGURE 27. The two were similar in appearance for the simple reason that Room Too Small Errors had a weighting of 100, and dominated the average cost. The extended flat region in FIGURE 25 shows that even once Room Too Small Errors were removed, "gaatt.c" took many generations to reach a cost of zero. This might be taken as an indication that Room Too Small errors were given too high a weighting in Test 5- they were not as hard to remove as the weightings would suggest.

The weightings used for the four types of hard constraint in test 5 were not tuned to provide optimal performance. From inspection of FIGURES 26 and 28 it appears that Related Class Clash Errors and Lecturer Double Booked Errors were not weighted highly enough. However, it is hard to know what the effects of such changes to constraint weightings would be, unless further tests are performed.

In order to find optimal weightings through empirical techniques a very large number of tests would need to be performed. A crude guide could be to use the average number of generations taken for each type of violation to be removed, as shown in Table 2. In this scheme the four types of error would have weightings of 55.1, 1270.6, 59.8 and 293.1 respectively. This method is crude because it fails to take into consideration interactions between the different constraints. Alternatively, many tests could be performed with different combinations of weightings in order to see which are best. A more advanced method would be to use the results of such a test to decide on which combinations are promising and then try other values in this region. Such a technique would be bordering on a metalevel optimisation strategy.

If sufficient information about the effects of hard constraint weightings was known then a weightings fine tuner could be incorporated into the GA. Ultimately, for optimal evolution, it could dynamically alter the weightings even as the algorithm was running. For example, if the number of occurrences of a particular type has become stagnant within the population then its weighting could be increased and vice versa. Tuning such as this might avoid the behaviour exhibited in FIGURE 26 and FIGURE 28 where the population managed to escape from regions where Related Class Clash Errors and Lecturer Double Booked Errors, respectively, had been eradicated. This would be a highly sophisticated metalevel optimisation technique. As such, its implementation is well beyond the scope of this paper.

Performance of "gaatt.c" in Test 6.

The optimal mutation rate was found to be at, or near, 16. This supports the earlier statement that mutation rates must be kept low (Davis, 1991) in order to maintain a balance between exploitation and exploration (Gen and Cheng, 1997). Mutation rate values larger than this or smaller than this provided slower evolution, in an asymmetric pattern.

With the mutation rate set to zero the cost of the population stagnated. This was almost definitely because, without mutation, no new alleles were introduced into the population. For high mutation rates the evolution did not stagnate, rather it progressed only very slowly. This could be attributed to the high mutation rate causing too much exploration of the (largely unpromising) search space and not enough exploitation of the genes found within the current population at any time.

It is important to note that the Mutation Rate Test only looked at the average cost after the first 500 generations. As such, it only gave direct information about the optimal mutation rate in the early stages of evolution. It is not, therefore known if some mutation rate other than 16 would provide faster evolution to a cost of 0. To record the time taken to evolve right down to a cost of 0 for a whole range of mutation rates would have taken many times longer to test, so the current test was considered adequate.

Ultimately, it is quite possible that optimal performance of the GA would require continual fine tuning of the mutation rate. The implementation of metalevel optimisations techniques such as this, go well beyond the aims of this thesis.

Performance of "gaatt.c" in Test 7.

The smallest possible population size (three) was seen to evolve quickest of all sizes tested in test seven. It is assumed that this is because "gaatt.c" was executed on a serial machine. As such, larger populations take more time in between each selection for extinction- so lower performance chromosomes are given more of a chance to breed.

On a parallel machine larger populations could be expected to perform better. It cannot be ruled out that even on the current hardware a different tuning of the GA parameters could allow larger populations to perform better than minimally small ones.

Possible Improvements to “gaatt.c”

Incorporating Soft Constraints into “gaatt.c”

As mentioned in the method section, soft constraints could be incorporated in the same way as hard constraints, though probably with smaller weightings. However, fast optimisation might require that some soft constraints be given large weightings, so as to ensure that selection pressure in their direction is sufficiently strong. In such a case, the difference between hard and soft constraints could be solely stored in the termination criteria. The GA could not terminate until all hard constraint errors were removed, though some soft constraint violations would remain. In any case, the architecture of “gaatt.c” is readily applicable to the inclusion of soft constraints.

Improving the existing Repair Strategy

Each time a timetable is created it must be repaired. As such, it is important that the repair strategy perform its task as efficiently as possible. The repair strategy requires two stages. In this implementation, they are separated so that the routine will not find an impasse if, when trying to allocate an unbooked class, there are no NULL bookings available. A smarter system might be able to perform the two steps of the repair strategy together. This could represent a significant improvement in the time taken to create each timetable. However, it would not reduce the number of timetables which are evaluated before a solution is found.

The Possible Incorporation of Further Repair Strategies on “gaatt.c”

As outlined in the theory section, the performance of a GA is improved as more repair strategies are included in it. In particular, it might be beneficial to create repair strategies which either minimise or eradicate instances of any particular type of hard constraint violation. This could drastically reduce the size of the solution space, and provide faster evolution.

Minimisation, or even removal of Room Too Small Errors appears to be a viable goal for a further repair strategy to deal with. This is true for two reasons. Firstly, because of all hard constraints violations considered in Tests 1 to 4 Room Too Small Errors took the largest number of generations to eradicate. Secondly, because existing theory of similar sorting problems might be readily applicable to their reduction. A system could be developed which sorts through a timetable ensuring that all classes booked to rooms too small for them are found suitable accommodation elsewhere. A less intelligent, though highly robust method could laboriously avoid the assigning of classes to rooms which are too small in the first place. Any repair strategy which minimises Room Too Small Errors would have the potential to greatly increase the speed of evolution.

Similarly, repair strategies could be found to minimise violations of any of the remaining types of hard constraints. Pre-existing theory relevant to any of these constraints, no matter how far removed from GAs, could be used as the basis for a repair strategy. The incorporation of further techniques for repair would be of greater importance if the magnitude of the problem was increased.

Initialisation of Timetables from Secondary Storage.

A further extension to “gaatt.c” would be to include the ability to initialise the population from a secondary source. This might be particularly useful, if it is intended that a timetable be kept as similar as possible from year to year. By choosing last year’s timetable as a basis for this year’s, such a goal might be achieved. However, that would not be the best way to perform such a task as the breadth of the search would be considerably reduced, and unbounded deviation from the start point could still occur. A more correct method of keeping timetables similar from year to year would be to base part of the evaluation of a proposed timetable on how similar it is to the timetable of the previous year. In such a case the population could still be initialised from a random point, thus ensuring a suitably broad search.

Consideration of Alternative Data Structures for Timetables.

A timetable can be represented in any of a number of ways. Each representation has advantages and disadvantages in terms of the performance of the GA. In particular, different timetable implementations will vary in how much space (ie. memory) they occupy and how easy they are to evaluate. Consequently they will vary in how complex they are to deal with and ultimately, how quickly they converge.

The advantage of the style chosen is that it automatically eradicates the chance of a classroom being double booked, thus removing one type of hard constraint violation. Other implementations would allow this type of hard constraint to be breached, but would remove the possibility of other hard constraint violations. For example, consider a timetable as an array of classes, where each class has stored with it a room number and a time. In this case, each class will automatically appear only once, but rooms could be double booked.

The disadvantage of the data structure chosen for timetables is that, because of the space wasted in NULL bookings, all timetables are as large as can be. However, in a resource tight timetable, there are less NULL bookings. Since "gaatt.c" was created specifically to deal with resource tight situations, the disadvantage of wasted space is not of tremendous importance.

Extending "gaatt.c" to the Complete Lecture Timetabling Problem.

In order to modify "gaatt.c" to handle the complete timetabling problem, further constraints must be incorporated within it. Generally this means that extensions must be made to the constraint data. Extensions to each type of constraint data will be discussed in turn.

A more advanced model for storing Lecturer information would contain a more detailed availability timetable which shows also preferred times for lecturing, preferred days for not lecturing and so on. The location of the lecturer's office or home room needs to be stored as a grid location (Cartesian coordinates), so that the approximate distance a lecturer needs to travel can be calculated. Furthermore, a lecturer's preferred rooms or room requirements might be stored (for example, wheelchair access might be required).

In a complete timetabling problem Classes would have more fields. There might be a field indicating the preferred location for a class. In the current system, each class can have only one lecturer- this could be expandable. The home room of each class would need to be stored as a grid location so that the proximity of each class to the home department could be assessed. Also, as mentioned previously, an extended system would allow each class to belong to more than one set of related classes.

More information could be stored about each Classroom. An availability timetable of the room would be stored (to take care of prior bookings, separate cleaning rosters etc) and a list of equipment which a room has (air conditioning, audio visual, laboratory equipment etc), which may be required/requested. The grid location of each classroom would need to be stored so that distances could be assessed.

The extension of the constraint data would mean that more types of constraint could be placed on the timetables. This could be expected to increase the time taken to find viable solutions. It would be reasonable to assume, for example, that doubling the number of constraints on the timetable would more than double the amount of time taken to reach a satisfactory result. However, the resultant timetable would be more applicable to real life situations. The addition of soft constraints, for example, would mean that rather than converging toward adequate timetables, the GA would converge toward excellent timetables. This flexibility of the method, whereby further constraints can be added without limit, means that almost any problem which a timetable might produce can be catered for.

In practice a Genetic Algorithm Automated Timetabling system would require an ergonomic front end, most probably of a graphical form. This would be used to enter and manipulate constraint data, and to extract proposed timetables. The operator of such a program could ultimately be a person with little or no knowledge of the underlying GA.

A Brief Discussion of Genetic Algorithms

The incorporation of intelligent methods- whether they are Lamarckian, Memetic, Repair strategies, or any other, into Genetic Algorithms can provide gestalt effects. The depth of one can be added to the breadth of the other, as the case may be. This ensures that a well constructed GA can always perform at least almost as well as any techniques which might rival it. This is true for the simple reason that a GA can incorporate its rivals within it, thus creating a hybridised GA.

Furthermore, unlike many of the techniques that humans have created for solving problems, GAs are readily applicable to highly parallel architectures. It is in this direction that GA are widely expected to excel.

Since their conception in 1975 GAs have undergone a great many changes. In particular they are no longer so closely comparable to biological evolution. The use of more complex data types means that in many cases the “genes” of the GA hardly resemble biological genes at all. Similarly, methods of mutation are no longer so closely paralleled with those in nature. Rather, any mutation methods which perform well on the given problem are used, regardless of a lack of biological counterpart. In ways such as this GA no longer imitate the specific methods of evolution observable in nature, rather, they encapsulate its essence.

Chapter 6: Conclusions.

It has been seen that “gaatt.c” performed directed evolution on a simplified Lecture Timetabling problem, and produced timetables void of hard constraint violations. These results were very promising.

The major improvement that could be made to its performance on the Simplified Lecture Timetabling Problem would be the further inclusion of Repair Strategies. Also, the fine tuning of its performance parameters could provide faster evolution.

The program was seen to be readily scalable to the Complete Lecture Timetabling Problem. This would be achieved by increasing the size of the University and by incorporating further constraints on the timetable.

GAs effectively demonstrated an ability to solve complex optimisation problems. Notably, this served to provide a very thorough introduction to the techniques employed and incorporated by Genetic Algorithms.

Appendix A: Reference List.

Anderson A and Simpson AR (1996): *Genetic Algorithm Optimisation Software in FORTRAN Research Report No. R136.* Department of Civil and Environmental Engineering, The University of Adelaide.

Buckles BP and Petry FE (1992): *Genetic Algorithms.* Los Alamitos : The IEEE Computer Society Press.

Burke EK, Newall JP and Weare RF(1995) : A Memetic Algorithm for University Exam Timetabling. In Burke E and Ross P (Eds): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 241-250.

Burke E and Ross P (Eds) (1996): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers .* New York: Springer-Verlag Berlin Heidelberg.

Carter MW and Laporte G (1995): Recent Developments in Practical Timetabling. In Burke E and Ross P (Eds): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 3-21.

Cooper TB and Kingston JH (1995): The Complexity of Timetable Construction Problems. In Burke E and Ross P (Eds): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 283-295.

Corne D and Ross P (1995): Peckish Initialisation Strategies for Evolutionary Timetabling. In Burke E and Ross P (Eds): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 227-240.

Davis L (Ed) (1991): *Handbook of Genetic Algorithms.* New York: Van Nostrand Reinhold.

de Werra D (1995): Some Combinatorial Models for Course Scheduling. In Burke E and Ross P (Eds): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 296-308.

Erben W and Keppler J (1995): A Genetic Algorithm Solving a Weekly Course-Timetabling Problem. In Burke E and Ross P (Eds): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 198-211.

Ergul A (1995): GA-based Examination Scheduling Experience at Middle East Technical University. In Burke E and Ross P (Eds): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 212-226.

Gell-Mann M (1994*): The Quark and the Jaguar - adventures in the simple and the complex.* London: Little, Brown and Company Limited.

Gen M and Cheng R (1997): *Genetic Algorithms and Engineering Design.* New York: John Wiley & Sons, Inc.

Goldberg DE (1989): Sizing Populations for Serial and Parallel Genetic Algorithms. In Buckles BP and Petry FE (1994): *Genetic Algorithms.* Los Alamitos: The IEEE Computer Society Press. pp 20-29.

Grefenstette J (1986): Optimisation of Control Parameters for Genetic Algorithms. In Buckles BP and Petry FE (1994): *Genetic Algorithms.* Los Alamitos : The IEEE Computer Society Press. pp 5-11.

Heitkoetter J (1993): *FAQ for comp.ai.genetic Usenet newsgroup*. Available from The Sante Fe Institute on the WWW at “http://alife.santafe.edu/~joke/encore/www/”

Hoener H (1996): *Genetic Programming Kernel*. Available on the WWW at "http://aif.wu-wien.ac.at/~geyers/archive/gpk/Dok/kurz/kurz.html"

NovaGenetica (1997): *Nova Genetica- Your #1 source for information on evolutionary algorithms*. Available on the WWW at "http://www.aracnet.com/~wwir/NovaGenetica"

Opcom (1996): *TIARA Timetabling and Room Allocation Software System FACT SHEET*. Toowong: Opcom.

Paechter B, Cumming A, Norman MG and Luchian H (1995): Extensions to a Memetic Timetabling System. In Burke E and Ross P (Eds): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 251-265.

Rankin RC (1995): Automatic Timetabling in Practice. In Burke E and Ross P (Eds): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 266-279.

Ravise C, Serbag M and Schoenauer M (1995): Induction-based Control of Genetic Algorithms. In Alliot J-M, Lutton E, Ronald E, Schoenauer M and Snyers D (Eds): *Lecture Notes in Computer Science 1063 Artificial Evolution European Conference, AE 95 Brest, France, September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 100-119.

Rich DC (1995): A Smart Genetic Algorithm for University Timetabling. In Burke E and Ross P (Eds): *Lecture Notes in Computer Science 1153 Practice and Theory of Automated Timetabling First International Conference, Edinburgh, U.K., August/September 1995, Selected Papers.* New York: Springer-Verlag Berlin Heidelberg. pp 181-197.

Shaffer R (1996): *Practical Guide to Genetic Algorithms.* Available on the WWW at "http://chem1.nrl/navy.mil/~shaffer/chemoweb.html"

Sherwood L (1993): *Human Physiology: from Cells to Systems* (2nd ed) USA: West Publishing Company.

Syswerda G (1991): Schedule Optimisation Using Genetic Algorithms. In Davis L (Ed): *Handbook of Genetic Algorithms.* New York: Van Nostrand Reinhold. pp 332-349.

University of Queensland (1997): *The University of Queensland General Class Timetable St Lucia Campus 1997*. St Lucia: The Academic Registrar, The University of Queensland.

Appendix B: C code for “gaatt.c”

```
/*
 * FILE:          gaatt.c
 *
 *
 * TOPIC:         Genetic Algorithm solution to lecture timetabling
 *                problems.
 *
 * METHOD:        Maintains a population of timetables. Timetables
 *                are bred, and evluated. Evaluation uses constraint
 *                data loaded from a file.
 *
 * CAVEAT:        The need for an excessive amount of nested statements
 *                makes this code particularly difficult to read.
 *                Pseudo code provided in the method section of the thesis
 *                "LECTURE TIMETABLING USING GENETIC ALGORITHMS"
 *                should help clarify the code.
 *
 * WRITTEN BY:           LEON BAMBIRICK, Jan 1997 - Oct 1997.
 *
 * STUDENT NO:           328 828 944.
 *
 */

#include <stdio.h>
#include <dos.h>
#include          <conio.h>
#include <math.h>
#include <stdlib.h>
#include <alloc.h>

#define max_population_size      3
#define max_num_of_lecturers     52
#define max_num_of_classes       202
#define max_room_num             15
#define days_in_week             5
#define hours_in_day             10

#define class_num                int



const   cost_of_related_class_clash      =      1;
const   cost_of_room_too_small           =    100;
const   cost_of_lecturer_double_booked   =    1;
const   cost_of_unavailable_lecturer     =    20;


/*
 *      Type declarations.
 */
typedef long cost_type;
typedef int coordinate_type;
typedef coordinate_type  room_location_type[2];
typedef char      class_name_type[6];
typedef int       associated_class_code_type;
typedef char        lecturer_name_type[30];
typedef int         lecturer_code_type;
typedef int         reference_num_of_lecturer_type;
typedef int         class_size_type;
typedef int         availability_time_table [days_in_week] [hours_in_day];
typedef struct lecturer {
        lecturer_name_type name_of_lecturer;
        lecturer_code_type code_of_lecturer;
        room_location_type home_office_location_of_lecturer;
        availability_time_table availability_time_table_of_lecturer;
};

struct collection_of_lecturers {
        struct lecturer  lecturers [max_num_of_lecturers];
        int num_of_lecturers;
} lecturer_constraints;

typedef struct class{
        class_name_type             name_of_class;
        reference_num_of_lecturer_type    reference_num_of_lecturer;
        class_size_type                   size_of_class;
        associated_class_code_type        associated_class_code;
        room_location_type                home_office_location_of_class;
};

struct collection_of_classes {
        struct class classes [max_num_of_classes];
        int num_of_classes;
        int num_of_class_associations;
} class_constraints;

typedef char  room_name_type[10];
typedef int   room_capacity_type;
typedef struct room{
        room_name_type                    name_of_room ;
        room_capacity_type                capacity_of_room;
        room_location_type                location_of_room;
        availability_time_table     availability_time_table_of_room;
};

struct collection_of_rooms {
        struct room rooms [max_room_num];
        int num_of_rooms;
} room_constraints;

typedef struct time_table {
        struct time_table   *next;
        class_num huge bookings [max_room_num] [days_in_week] [hours_in_day];
        cost_type       cost;
        cost_type       rcc_error_count;
        cost_type       rts_error_count;
        cost_type       ldb_error_count;
        cost_type       lua_error_count;
};

typedef int population_size_type;
typedef char colony_name_type[80];
struct colony {
        colony_name_type                name_of_colony;
        struct time_table           *first_time_table;
        struct time_table           *last_time_table;
        population_size_type            population_size;
        cost_type average_cost;
        cost_type       rcc_error_count;
        cost_type       rts_error_count;
        cost_type       ldb_error_count;
        cost_type       lua_error_count;
} solution_colony;

typedef char file_name_type[12];

/*
 *  Prototypes
 */
int main(void);
int initialise_constraints(file_name_type      constraint_file_name);
int repair_strategy(struct time_table  *curr_ptr);
int repair_strategy_0(struct time_table  *curr_ptr);
int calculate_cost(struct time_table  *curr_ptr);
int initialise_colony(void);
int kill_costly_colony_members(void);
int breed_colony(void);
int output_colony(file_name_type out_file_name);
int mutate(struct time_table  *curr_ptr);
int pop_size(void);
int population_size;
long related_classes(struct time_table  *curr_ptr);
long room_too_small(struct time_table * curr_ptr);
long lecturer_unavailable(struct time_table * curr_ptr);
int fget_string(FILE *in, char wordy[80]);
int value_of(char wordy[80]);
int fget_line_value(FILE *in);
int fgetc_value(FILE *in);
void fputn(int number, FILE *fp);

/*
 *  Global variables
 */
int mutation_rate =         16;
int num_of_trials = 0;

/*
 *  main -
 *
 *  evolves timetables in the direction of minimum number of breaches
 *  of all constraints included.
 *
 *  Terminates when one perfect timetable has been created.
 *
 *
 */
int main(void)
{
        int generations = 0;
        int num_of_generation = 0;
        cost_type   maximum_allowed_cost = 0;
```

```
char        constraint_file_name[] = "c:\\tcc\\thesis\\constr3.ctr";
char        out_file_name[] = "c:\\tcc\\thesis\\out.pop";

if (initialise_constraints(constraint_file_name)!=0)
            exit(1);
randomize();
population_size = max_population_size;
if (initialise_colony()!=0)
            exit(1);
if (find_average_cost()!=0)
            exit(1);

/*
 * repeat until a perfect time table is found.
 */
while (solution_colony.first_time_table -> cost
            > maximum_allowed_cost)            {

            /*
             * kill off the costliest half of the population
             */

            if (kill_costly_colony_members()!=0)
                        exit(1);

            /*
             * find the average cost of the population.
             */
            if (find_average_cost()!=0)
                        exit(1);

            /*
             * breed the population back up to full size.
             */
            while (solution_colony.population_size <
                                    max_population_size){
                        num_of_trials= num_of_trials +1;
                        if (breed_colony()!=0)
                                    exit(1);
            }
            num_of_generation++;
            /*
             *          Output the current status to the screen.
             */
            printf("  %d ",(solution_colony.average_cost));
            printf("  %d ",num_of_trials);
            printf("  %d ",num_of_generation);
            printf("  (%d) \n",
                        solution_colony.first_time_table -> cost);
            num_of_trials = 0;
}

/*
 *  output the colony to a ".pop" text file
 */
if (output_colony(out_file_name) !=0)
            exit(1);
printf("\nfinished after %d generations.\n",generations);
```

```
        return(0);
}

/*
 * fget_string-
 *
 * used for getting a string from a file
 *
 */
int fget_string(FILE *in, char wordy[80])
{
        int prevcha = 'a';
        int count = 0;

        while ((prevcha != '\n') && (!feof(in))) {
                prevcha = fgetc(in);
                if (prevcha != '\n') {
                        wordy[count]=prevcha;
                        count++;
                }
        }
        wordy[count] = NULL;
        return 0;
}

/*
 * value_of-
 *
 * returns the integer values of a string.
 */
int value_of(char wordy[80])
{

        int a;
        int count = 0;
        int val = 0;
        int row = 0;

        while(wordy[count] != NULL) count++;
        for (a=0; a<count; a++){
                val += (wordy[count-a-1]-'0') *pow(10, row);
                row++;
        }
        return val;
}

int fget_line_value(FILE *in)
{
        char wordy[80];

        fget_string(in, wordy);
        return  value_of(wordy);

}

int fgetc_value(FILE *in)
{
        char ch;
        ch = fgetc(in);
        return (ch - '0');
}

/*
 * initialise_constraints-
 *
 *
 * PRE: the file references by file_name_type contains valid constraint
 *      data
 *
 * POST: All class constraints, lecturer constraints and room
 *    constraints are loaded from the file.
 *
 */
int initialise_constraints(file_name_type    constraint_file_name)
{
        FILE *in;
        char wordy[80];
        char prevcha;
        int a = 0;
        int day, hour;

        if ((in = fopen(constraint_file_name, "rt")) == NULL)
        {
                fprintf(stderr, "Cannot open input file.\n");
                return 1;
        }
        fget_string(in,wordy);
        fget_string(in,wordy);
        class_constraints.num_of_classes = fget_line_value(in);
        class_constraints.num_of_class_associations = fget_line_value(in);
        /*
         * read all data about class constraints from the file.
         */
        for (a = 1; a <= ((class_constraints.num_of_classes)); a++) {
                fget_string(in,class_constraints.classes[a].name_of_class);
                class_constraints.classes[a].reference_num_of_lecturer
                        = fget_line_value(in);
                class_constraints.classes[a].size_of_class
                        = fget_line_value(in);
                class_constraints.classes[a].home_office_location_of_class[1]
                        = fget_line_value(in);
                class_constraints.classes[a].home_office_location_of_class[2]
                        = fget_line_value(in);
                class_constraints.classes[a].associated_class_code
                        = fget_line_value(in);
        }
        class_constraints.classes[0].reference_num_of_lecturer = 0;
        class_constraints.classes[0].size_of_class = 0;
        class_constraints.classes[0].associated_class_code = 0;
        lecturer_constraints.num_of_lecturers = fget_line_value(in);
        for (a = 0; a < lecturer_constraints.num_of_lecturers; a++) {
                fget_string(in,lecturer_constraints.
                        lecturers[a].name_of_lecturer);

                lecturer_constraints.lecturers[a].
```

```
                  home_office_location_of_lecturer[1]
                        = fget_line_value(in);

            lecturer_constraints.lecturers[a].
                  home_office_location_of_lecturer[2]
                        = fget_line_value(in);

            for (day = 0; day < days_in_week; day++) {
                  for (hour=0; hour<hours_in_day; hour++) {

                        lecturer_constraints.lecturers[a].
                              availability_time_table_of_lecturer
                              [day][hour]
                        = fgetc_value(in);
                  }
                  fget_string(in,wordy);
            }
      }
      room_constraints.num_of_rooms = fget_line_value(in);
      for (a = 0; a<room_constraints.num_of_rooms; a++) {
            fget_string(in, room_constraints.rooms[a].name_of_room);
            room_constraints.rooms[a].capacity_of_room =
                  fget_line_value(in);
            room_constraints.rooms[a].location_of_room[1] =
                  fget_line_value(in);
            room_constraints.rooms[a].location_of_room[2] =
                  fget_line_value(in);
            for (day = 0; day < days_in_week; day++) {
                  for (hour = 0; hour<hours_in_day; hour++) {
                        room_constraints.rooms[a].
                        availability_time_table_of_room[day][hour]
                        = fgetc_value(in);
                  }
                  fget_string(in,wordy);
            }
      }
      fclose(in);
      return(0);
}


/*
 * repair_strategy-
 *
 * Performs the second stage of repair to the timetable pointed to
 * by curr_ptr
 *
 * PRE: Each class is booked zero or more times
 *
 * POST: Each class is booked either zero times or one time.
 *
 */
int repair_strategy(struct time_table  *curr_ptr)
{
/*

 PRE: Each class can appear 0 times, 1 time or 2 times in a timetable
 POSSIBLE PRE: Maybe a class can appear more than twice- two reasons:
```

```
            may want to have more than two parents. May help in making
            this a simple "random population" high cost time table generator.

 POST: It must be true that for each class there is one and only one
 booking for it in a week.

  Any class which is not booked exactly once must be corrected.

 A suggested style for doing this is as follows:
 */



struct booking_location {
            int which_room;
            int which_day;
            int which_hour;
            struct booking_location  *next;
};

            struct booking_location  *first;
            struct booking_location  *curr_booking;
            struct booking_location  *booking_to_remove;
            int cur_class;
            int class_occurred;
            int cur_room;
            int day;
            int hour;
            int one_to_remove;
            char chr;
            int a = 0;

            for (cur_class=1; cur_class<max_num_of_classes; cur_class++) {
                        class_occurred = 0;
                        for (cur_room = 0; cur_room<max_room_num;cur_room++)
                                    for (day = 0; day<days_in_week; day++)
                                                for (hour = 0; hour<hours_in_day; hour++){
                                                            if (curr_ptr -> bookings
                                                              [cur_room][day][hour] ==
                                                              cur_class) {
                                                                        class_occurred++;

if (class_occurred == 1) {
            if (( (first) =
            (struct booking_location *)malloc(sizeof(struct booking_location)))
            == NULL) {
                        printf("\n insufficient memory for this booking\n");
                        exit(1);
            }
            curr_booking = first;
}
else {
            if (( (curr_booking->next) =
            (struct booking_location  *)malloc(sizeof(struct booking_location)))
            == NULL) {
                        printf("\n insufficient memory for booking allocation \n");
                        exit(1);
            }
            curr_booking = curr_booking -> next;
}
curr_booking -> which_room = cur_room;
curr_booking -> which_day = day;
curr_booking -> which_hour = hour;


                        }
            }
            if (class_occurred == 1) {
                        free(first);
            } else
                        if (class_occurred > 1 ) {
                                    while (class_occurred > 1) {
                                                curr_booking = first;
                                                one_to_remove = random(class_occurred);
                                                for (a=0;a<one_to_remove;a++) {
                                                            curr_booking = curr_booking -> next;
                                                }
                                    cur_room = curr_booking -> which_room;
                                    day = curr_booking -> which_day;
                                    hour = curr_booking -> which_hour;
                                    if (curr_ptr -> bookings [cur_room] [day] [hour]
                                                == cur_class) {
                                    curr_ptr -> bookings [cur_room] [day] [hour] = 0;
                        if (one_to_remove == 0) {
                                    booking_to_remove = first;
                                    first = first -> next;
                        }
                        else
                        if (one_to_remove != class_occurred)
                          {
                                    curr_booking = first;
                                    for (a=0; a<(one_to_remove-1); a++){
                                    curr_booking = curr_booking -> next;
                                    }
                                    booking_to_remove = curr_booking -> next;
                                    curr_booking -> next = (curr_booking -> next)-> next;
                          }
                        free(booking_to_remove);
                        class_occurred --;
               }
                        free(first);
                                    }
                        }
            }
            return(0);
}

/*
 * repair_strategy_0-
 *
 * Performs the second stage of repair to the timetable pointed to
 * by curr_ptr
 *
 * PRE: Each class is booked either zero times or one time
 *
 * POST: Each class is booked precisely once.
 *
 */
int repair_strategy_0(struct time_table  *curr_ptr)
{
            int cur_class;
            int class_occurred;
            int cur_room;
            int day;
            int hour;
 int one_to_remove;
 char chr;

 for (cur_class=1; cur_class<class_constraints.num_of_classes; cur_class++) {
     class_occurred = 0;
     for (cur_room = 0; cur_room<max_room_num;cur_room++)
                 for (day = 0; day<days_in_week; day++)
                                   for (hour = 0; hour<hours_in_day; hour++) {
                                               if (curr_ptr ->
                                                           bookings [cur_room][day][hour]
                                                           == cur_class) {
                                                           class_occurred++;
                                               }
                                   }
            if (class_occurred == 0) {
                        while (class_occurred == 0) {
                                   cur_room = random(max_room_num);
                                   day  = random(days_in_week);
                                   hour = random(hours_in_day);

                                   if (curr_ptr ->
                                               bookings [cur_room] [day] [hour] == 0) {

                                               curr_ptr ->
                                                           bookings [cur_room] [day] [hour]
                                                                       = cur_class;
                                               class_occurred ++;
                                               }
                                   }
                        }
            }
            return(0);
}


/*
 * related classes-
 *
 * counts the number of times that related classes are booked
 * at the same time in the timetable pointed to by curr_ptr
 *
 */
long related_classes(struct time_table  *curr_ptr)
{
            int num_of_occurrences=0;
            int curr_class_group = 0;
            int this_group_has_occurred = 0;
            int curr_room;
            int curr_day;
```

```
        int hour;
        int this_class_group;
        int this_class_num;

        for (curr_class_group = 1;
                curr_class_group <=
                class_constraints.num_of_class_associations;
                curr_class_group ++) {

                for (curr_day=0; curr_day < days_in_week; curr_day++) {
                        for (hour=0; hour< hours_in_day; hour++) {
                                this_group_has_occurred = 0;
                                for (curr_room = 0;
                                        curr_room < max_room_num;
                                        curr_room ++) {

                                                this_class_num =
                                                (curr_ptr ->
                                                bookings
                                                [curr_room][curr_day][hour]);

                                                if (this_class_num!=0) {

                                                   this_class_group =
                                                   class_constraints.
                                                   classes
                                                   [this_class_num].
                                                   associated_class_code;
                                                   if
                                                   (this_class_group
                                                   == curr_class_group) {

                                                this_group_has_occurred ++;


                                                        }
                                                }
                                        }
                                if (this_group_has_occurred>1) {
                                        num_of_occurrences +=
                                                (this_group_has_occurred-1);
                                }
                        }
                }
        }
        return num_of_occurrences;
}

/*
 * room_too_small-
 *
 * counts the number of times a class is booked to a room which is
 * too small for it in the timetable pointed to by curr_ptr
 *
 */
long room_too_small(struct time_table  *curr_ptr)
{
        int num_of_occurrences=0;
        int curr_room;
        class_size_type curr_size_available;
        class_size_type curr_size_allocated;
        int curr_day;
        class_num  curr_class;
        int hour;

        for (curr_room=0;
           curr_room < room_constraints.num_of_rooms; curr_room ++){

                curr_size_available =
                        room_constraints.rooms[curr_room].capacity_of_room;

                for (curr_day=0; curr_day < days_in_week; curr_day++)
                        for (hour=0; hour< hours_in_day; hour++){

                                curr_class = curr_ptr ->
                                bookings [curr_room][curr_day][hour];

                                if (curr_class != 0) {
                                        curr_size_allocated =
                                        class_constraints.
                                        classes[curr_class].
                                        size_of_class;

                                        if (curr_size_available
                                          < curr_size_allocated) {
                                                num_of_occurrences ++;
                                        }
                                }
                        }
        }
        return num_of_occurrences;
}


/*
 * lecturer_double_booked-
 *
 * counts the number of times a lecturer is double booked in the
 * timetable pointed to by curr_ptr
 *
 */

long lecturer_double_booked(struct time_table  *curr_ptr)
{
        int num_of_occurrences=0;
        int curr_room;
        int lecturer_num;
        int curr_lecturer;
        int num_of_bookings_at_this_time;
        int current_lecturer_is;
        struct class current_class_is;
        int curr_class;
        int curr_class_num;
        int curr_day;
        int hour;
	for (lecturer_num = 0; lecturer_num < lecturer_constraints.
				num_of_lecturers; lecturer_num++) {
		for (curr_day = 0; curr_day < days_in_week; curr_day++)
			for (hour = 0; hour< hours_in_day; hour++){
				num_of_bookings_at_this_time = 0;
				for (curr_room = 0;
				    curr_room < max_room_num;
				    curr_room ++){

					curr_class_num =
						(curr_ptr ->
						bookings
						[curr_room][curr_day][hour]);
					if (curr_class_num!=0){
						curr_lecturer =
						class_constraints.
						classes[curr_class_num].
						reference_num_of_lecturer;

						if (curr_class_num != 0) {
							if (lecturer_num ==
							curr_lecturer) {

					 num_of_bookings_at_this_time++;


							}
						}
					}
					if (num_of_bookings_at_this_time>1) {

					num_of_occurrences
					+=(num_of_bookings_at_this_time-1);

					}
				}
			}
	}
	return num_of_occurrences;
}

/*
 * lecturer_unavailable -
 *
 * returns the number of violations of the constraint
 * "Lecturer's cannot be booked when they have prior commitments."
 * in the timetable pointed to by curr_ptr.
 *
 */
long lecturer_unavailable(struct time_table  *curr_ptr)
{
	int num_of_occurrences=0;
	int curr_room;
	int lecturer_num;
	int curr_lecturer;
	int current_lecturer_is;
	struct class current_class_is;
	int curr_class;
	int curr_class_num;
	int curr_day;
	int hour;

	for (curr_day=0; curr_day < days_in_week; curr_day++)
		for (hour=0; hour< hours_in_day; hour++){
			for (curr_room=0; curr_room
				< max_room_num; curr_room ++){

				curr_class_num
					= (curr_ptr -> bookings
					[curr_room][curr_day][hour]);

				if (curr_class_num != 0) {
					curr_lecturer =
						class_constraints.
						classes[curr_class_num].
						reference_num_of_lecturer;

					if (lecturer_constraints.
						lecturers[curr_lecturer].
					availability_time_table_of_lecturer
					[curr_day][hour] == 0) {
						num_of_occurrences++;
					}
				}
			}
		}
	return num_of_occurrences;
}

/*
 * calculate_cost-
 *
 * calculate_cost determines the cost of the timetable pointed to
 * by curr_ptr. The number of violations of each constraint is
 * multiplied by the weighting for that constraint and added to
 * the total.
 *
 */
int calculate_cost(struct time_table  *curr_ptr)
{
	cost_type the_cost = 0;
	cost_type problem1;
	cost_type problem2;
	cost_type problem3;
	cost_type problem4;
	struct time_table *spare_ptr;

	spare_ptr = curr_ptr;
	spare_ptr -> cost = 0;
	curr_ptr -> rcc_error_count = related_classes(curr_ptr);
	curr_ptr -> rts_error_count = room_too_small(curr_ptr);
	curr_ptr -> ldb_error_count = lecturer_double_booked(curr_ptr);
	curr_ptr -> lua_error_count = lecturer_unavailable(curr_ptr);
	problem1 = (curr_ptr ->
		rcc_error_count * cost_of_related_class_clash);
	problem2 = (curr_ptr ->
		rts_error_count * cost_of_room_too_small);
	problem3 = (curr_ptr ->
		ldb_error_count * cost_of_lecturer_double_booked);
	problem4 = (curr_ptr ->
		lua_error_count * cost_of_unavailable_lecturer);
	printf("              ");
	printf(" %4d", problem1);
	printf(" %5d", problem2);
	printf(" %5d", problem3);
	printf(" %5d ", problem4);
	the_cost = problem1 + problem2 + problem3 + problem4;
	spare_ptr -> cost = the_cost;
	return(0);
}



/*
 * fputn -
 *
 * used for writing an integer from 0 to 999 into a file.
 *
 */
void fputn(int a, FILE *fp)
{
	int b;
	b=a;
	if (b>99) {
		b = b % 100;
		b = (a-b) / 100;
		fputc(b+48, fp);
		b = a - (b*100);
	} else
		fputc(48, fp);
	if (b>9) {
		a = b;
		b = b % 10;
		b = (a - b) / 10 ;
		fputc(b+48, fp);
		b = a - (b*10);
	} else
		fputc(48, fp);
	fputc(b+48, fp);
	fputc('\n', fp);
}


/*
 * initialise_colony -
 *
 * creates a random population by repairing NULL timetables.
 *
 */
int initialise_colony(void)
{
	struct time_table  *curr_ptr;
	struct time_table  *lagging_ptr;
struct time_table  *test_tube;
char wordy[80];
int solution, cur_room,day,hour;
int final_population_size;

solution_colony.population_size = 0;
final_population_size = population_size;
while (solution_colony.population_size < final_population_size) {
        if (( (test_tube) =
        (struct time_table  *)malloc(sizeof(struct time_table)))
        == NULL) {
                printf("\ninsufficient memory for TT allocation \n");
                exit(1);
        }
        if (test_tube -> next != NULL) {
                test_tube -> next = NULL;
        }
        if (solution_colony.last_time_table -> next != NULL ) {
                solution_colony.last_time_table -> next = NULL;
        }
for (cur_room = 0; cur_room < max_room_num;cur_room++)
        for (day = 0; day < days_in_week; day++)
                for (hour = 0; hour < hours_in_day; hour++)
                        (test_tube ->
                                bookings[cur_room][day][hour]) = 0;
repair_strategy_0(test_tube);
calculate_cost(test_tube);
if (solution_colony.population_size == 0) {
        solution_colony.first_time_table = test_tube;
        solution_colony.last_time_table = test_tube;
        solution_colony.population_size++;
} else {
        curr_ptr = solution_colony.first_time_table;
        if (curr_ptr -> cost  >= test_tube -> cost) {
                test_tube -> next = curr_ptr;
                solution_colony.first_time_table = test_tube;
                solution_colony.population_size++;
        } else {
                curr_ptr = solution_colony.first_time_table;
                while ((curr_ptr-> next-> cost <= test_tube -> cost)
                        && (curr_ptr -> next != NULL) ){
                                if (curr_ptr -> next != NULL) {
                                        curr_ptr = curr_ptr -> next;
                                }
                        };
                        if (curr_ptr ==
                                solution_colony.last_time_table) {
                                solution_colony.last_time_table
                                        = test_tube;
                                test_tube -> next = NULL;
                        } else  test_tube -> next = curr_ptr -> next;
                        curr_ptr->next = test_tube;
                        solution_colony.population_size++;
                }
        }
        solution_colony.last_time_table -> next = NULL;
        curr_ptr = solution_colony.first_time_table;
        while (curr_ptr != NULL) {
```

```
                        curr_ptr = curr_ptr -> next;
                }
        test_tube = NULL;
        free(test_tube);
        }
 return(0);
}

/*
 * find_average_cost -
 *
 * Finds mean of the cost of each timetable
 * The result is entered in the colonies average cost field.
 *
 *
 */
int find_average_cost()
{
        cost_type  sum_of_costs = 0;
        cost_type  sum_of_error1 = 0;
        cost_type  sum_of_error2 = 0;
        cost_type  sum_of_error3 = 0;
        cost_type  sum_of_error4 = 0;
        struct time_table  *curr_ptr;
        long a;

        curr_ptr = solution_colony.first_time_table;
        a=0;
        while (curr_ptr != NULL) {
                a++;
                sum_of_costs += curr_ptr -> cost;
                sum_of_error1 += curr_ptr -> rcc_error_count;
                sum_of_error2 += curr_ptr -> rts_error_count;
                sum_of_error3 += curr_ptr -> ldb_error_count;
                sum_of_error4 += curr_ptr -> lua_error_count;
                curr_ptr = curr_ptr-> next;
        }
        solution_colony.average_cost =
                (sum_of_costs / (solution_colony.population_size));
        solution_colony.rcc_error_count =
                (sum_of_error1 / (solution_colony.population_size));
        solution_colony.rts_error_count =
                (sum_of_error2 / (solution_colony.population_size));
        solution_colony.ldb_error_count =
                (sum_of_error3 / (solution_colony.population_size));
        solution_colony.lua_error_count =
                (sum_of_error4 / (solution_colony.population_size));
        return(0);
}

typedef struct colony_cost_statistics {
        cost_type costs[max_population_size];
        cost_type average_cost;
        cost_type median_cost;
        cost_type fatal_cost;
};
```

```
/*
 * kill_costly_colony_members -
 *
 * The colony is already order from least costly to most costly.
 * Decide what number of people to kill (x).
 * find the (pop-x)th creature. Make them the "last" in line.
 * free each creature after that.
 *
 */

int kill_costly_colony_members()
{
	int amount_to_kill;
	float kill_ratio = 0.5; //need not be integer.
	struct time_table  *curr_ptr;
	struct time_table  *lagging_ptr;
	int a;

	amount_to_kill = (solution_colony.population_size * kill_ratio);
	while (solution_colony.population_size-amount_to_kill < 2) {
		amount_to_kill--;
	}
	curr_ptr = solution_colony.first_time_table;
	for (a=0; a<(solution_colony.population_size-amount_to_kill)-1;a++){
		curr_ptr = curr_ptr ->next;
	}
	solution_colony.last_time_table = curr_ptr;
	curr_ptr = curr_ptr -> next;
	solution_colony.last_time_table -> next = NULL;
	do {
		lagging_ptr = curr_ptr;
		curr_ptr = curr_ptr -> next;
		free(lagging_ptr);
		solution_colony.population_size--;
	} while (curr_ptr != NULL);
	return(0);
}

/*
 * pop_size -
 *
 * used for verifying that the population's size is correctly recorded.
 */
int pop_size()
{
	int a=0;
	struct time_table  *curr_ptr;

	curr_ptr = solution_colony.first_time_table;
	while (curr_ptr != NULL) {
		a++;
		curr_ptr = curr_ptr -> next;
	}
	return a;
}


/*
```

```
 * mutate -
 *
 * Performs mutations on a child.
 * Probability that a gene will undergo mutation is (2*mutation_rate)/1000
 * Method of mutation is to randomly switch any two genes.
 *
 */
int mutate(struct time_table  *curr_ptr)
{
        class_num temporary;
        int cur_room;
        int cur_day;
        int cur_hour;
        int random_room;
        int random_day;
        int random_hour;


        for (cur_room = 0; cur_room<max_room_num;cur_room++)
                for (cur_day = 0; cur_day<days_in_week; cur_day++)
                        for (cur_hour = 0;
                                cur_hour < hours_in_day; cur_hour++){

                                if (random(1000)<mutation_rate) {


                                        temporary =
                                                curr_ptr -> bookings
                                                [cur_room] [cur_day]
                                                [cur_hour];

                                        random_room = random(max_room_num);
                                        random_day = random(days_in_week);
                                        random_hour = random(hours_in_day);

                                        curr_ptr -> bookings
                                        [cur_room][cur_day]
                                        [cur_hour] =
                                        curr_ptr -> bookings
                                        [random_room][random_day]
                                        [random_hour];

                                        curr_ptr -> bookings
                                        [random_room][random_day]
                                        [random_hour]=
                                        temporary;
                                }
                        }
        return(0);
}


/*
 * breed_colony -
 *
 * selects two members of the population, at random, to act as
 * parents.
 * Their genetic makeup using cross over, to produce a child
 * The child is mutated, repaired and evaluated before being placed
 * in the populatio.
 *
 */
int breed_colony()
{
        int cross_over_rate = 2;
        struct time_table  *mother;
        struct time_table  *father;
        struct time_table  *test_tube;
        struct time_table  *curr_ptr;
        int mother_pos;
        int father_pos;
        int a;
        int cur_room;
        int day;
        int hour;
        int state=0;
        printf("* %3d", num_of_trials);
        mother_pos = random(solution_colony.population_size);
        do {
                father_pos = random(solution_colony.population_size);
        }
        while (mother_pos == father_pos);
        mother = solution_colony.first_time_table;
        for (a=0; a<mother_pos; a++){
                mother = mother -> next;
        }
        father = solution_colony.first_time_table;
        for (a=0; a<father_pos; a++){
                father = father -> next;
        }
        if (( test_tube = (struct time_table
                *)malloc(sizeof(struct time_table))) == NULL) {
                printf("\n insufficient memory for TT \n");
                exit(1);
        }
        state=random(2)+1;
        for (day = 0; day<days_in_week; day++) {
                for (hour = 0; hour<hours_in_day; hour++) {
                        for (cur_room = 0; cur_room<max_room_num;cur_room++)
                        {
                                if (state==1) {

                                        test_tube ->
                                        bookings [cur_room] [day] [hour] =
                                        mother ->
                                        bookings [cur_room] [day] [hour];


                                        if (random(cross_over_rate)==0) {
                                                state = 2;
                                        }
                                } else {
                                        test_tube ->
                                        bookings [cur_room] [day] [hour] =

                                        father ->
```

```
                                                  bookings [cur_room] [day] [hour];
                                                  if (random(cross_over_rate)==0) {
                                                          state = 1;

                                                  }
                                          }
                                  }
                          }
                  }
                  mutate(test_tube);
                  repair_strategy(test_tube);
                  repair_strategy_0(test_tube);
                  calculate_cost(test_tube);
                  {
                          printf("      .....cost= %8d\n", test_tube -> cost);
                          curr_ptr = solution_colony.first_time_table;
                          if (curr_ptr -> cost  >= test_tube -> cost) {
                                  test_tube -> next = curr_ptr;
                                  solution_colony.first_time_table = test_tube;
                                  solution_colony.population_size++;
                          } else {
                                  curr_ptr = solution_colony.first_time_table;
                                  while ((curr_ptr-> next-> cost <= test_tube -> cost)
                                          && (curr_ptr -> next != NULL) ){
                                          if (curr_ptr -> next != NULL) {
                                                  curr_ptr = curr_ptr -> next;
                                          }
                                  };
                                  if (curr_ptr == solution_colony.last_time_table) {
                                          solution_colony.last_time_table = test_tube;
                                          test_tube -> next = NULL;
                                  } else     test_tube -> next = curr_ptr -> next;
                                  curr_ptr->next = test_tube;
                                  solution_colony.population_size++;
                          }
                  }
                  return(0);
          }

          /*
           * output_colony -
           *
           * Outputs all genes of all creatures to a ".pop" text file.
           * They can be later retrieved and used as an initialisation point.
           *
           */
          int output_colony(file_name_type out_file_name)
          {
                  FILE *out;
                  struct time_table  *curr_ptr;
                  int cur_room, day, hour;

                  if ((out = fopen(out_file_name, "w")) == NULL) {
                          fprintf(stderr, "Cannot open output file.\n");
                          return 1;
                  }
                  curr_ptr = solution_colony.first_time_table;
                  while ((curr_ptr->next)  != NULL) {
```

```
				for (cur_room = 0; cur_room<max_room_num;cur_room++)
					for (day = 0; day<days_in_week; day++)
						for (hour = 0; hour<hours_in_day; hour++){
							fputn(curr_ptr -> bookings
							[cur_room][day][hour],out);
						}
				curr_ptr = (curr_ptr -> next);
			}
			fclose(out);
			return(0);
		}
```