

An Application of Genetic Algorithms to University Timetabling

BSc (Hons) Computer Science
Robert Gordon University, Aberdeen

Author: Alexander Brownlee

Project Supervisor: Dr. John McCall

Date: 29/04/2005

Abstract

Timetabling is a highly complex problem which is part of the wider field of scheduling, a subject of extensive research over the past half-century. Scheduling is broadly defined as “the problem of the allocation of resources over time to perform a set of tasks” [1] and is a prominent example of a set of notoriously difficult NP-hard, constrained, combinatorial optimisation problems.

There are several different sub-categories of timetabling; within a university setting (the scenario for which sample data is readily available to this project) it can be divided into the distinctly different problems of timetabling the exam diet and normal class delivery. This project conducts an investigation into the problem of class timetabling and attempts to reproduce three different approaches to solving it. Being classed as NP-hard, no deterministic algorithm can be devised to generate a timetable within a reasonable time. This problem is a good candidate for the use of genetic algorithms (GAs); these will be examined in detail before proceeding to a detailed analysis of timetabling and the application of two different GAs to it. An extension of genetic algorithms known as memetic algorithms is also investigated and applied to the problem. Following this, the algorithms are optimised and a comparison is made of them.

Keywords

Chromosome	Crossover	Elitism	Evolution
Fitness	Fractional Factorial	Generation	Genetic Algorithm
Greedy Algorithm	Java	Local Search	Memetic Algorithm
Mutation	NP-Hard	Object-Oriented	Population
Response Surface	Scheduling	Selection	Timetabling

Declaration

I confirm that the work contained in this report has been composed solely by myself. All sources of information have been specifically acknowledged and verbatim extracts are distinguished by quotation marks.

Contents

Abstract.....	2
Declaration.....	3
Contents.....	4
Figures.....	6
Tables.....	6
1. Introduction.....	7
1.1. Manual timetabling.....	7
1.2. Alternative Approaches to the Timetabling Problem.....	8
1.2.1. Overview.....	8
1.2.2. Tabu Search.....	8
1.2.3. Tiling Algorithms.....	9
1.2.4. Simulated Annealing.....	9
1.2.5. Agents.....	10
1.2.6. Why Choose GAs Over the Other Approaches?.....	10
2. Background Theory.....	11
2.1. Genetic Algorithms.....	11
2.1.1. History and General Principals.....	11
2.1.2. Selection, Elitism and Steady State GAs.....	12
2.1.3. Crossover / Recombination.....	15
2.1.4. Mutation.....	19
2.2. Memetic Algorithms.....	20
3. GAs and MAs Applied to Timetabling.....	22
3.1. GAs and Timetabling.....	22
3.2. MAs and Timetabling.....	23
4. Practical Implementation.....	26
4.1. Practicalities of the Problem.....	26
4.1.1. Constraints.....	26
4.2. Class Structure.....	28
4.2.1. The Requirements and Timeslots classes.....	28
4.2.2. The Timetable class.....	30
4.2.3. TimetableGUI class.....	33
4.2.4. GA and MA Classes.....	33
4.2.5. GA Operators.....	35
4.2.6. Greedy algorithm.....	37
4.3. Reading the data.....	37
4.4. Problems with Algorithm Speed.....	39
4.4.1. Overview.....	39
4.4.2. Sorting the Modules.....	40
4.4.3. Boltzmann Selection.....	41
4.4.4. The Global Fitness Function.....	42
4.4.5. The Local Fitness Function.....	43
5. Optimisation of the Algorithms.....	47
5.1. Overview.....	47
5.2. Fractional Factorial Screening Experiment.....	47
5.2.1. Fractional Factorial Analysis.....	47
5.2.2. Factors.....	48
5.2.3. Approach Taken.....	50
5.2.4. Results.....	51
5.3. Response Surface Experiment.....	53
5.3.1. Summary.....	53
5.3.2. Results.....	54
5.4. Confirmation Experiment.....	56
6. Comparison of the Algorithms.....	58

6.1. Experiments.....	58
6.2. Results.....	59
6.3. Manually Generated Timetable.....	61
7. Conclusions and Future Work.....	63
8. References.....	66
8.1. Books / Research Papers.....	66
8.2. World Wide Web URLs.....	67
8.3. Presentation.....	67
Appendix A. Glossary.....	68
Appendix B. Pseudocode.....	70
B.1. Local search.....	70
B.2. Local fitness.....	70
B.3. Fitness.....	71
B.4. Number of Clashes.....	71
B.5. Greedy room allocator.....	72
B.6. Copy alleles to timetable.....	72
B.7. Crossover.....	72
B.8. Number Creep Mutation.....	73
B.9. Tournament Selection.....	73
B.10. Boltzmann Selection.....	73
Appendix C. Class Diagrams.....	75
C.1. Timetable Classes.....	75
C.2. GA Classes.....	76
Appendix D. Sample Timetables.....	77
Appendix E. Data from Experiments.....	78
E.1. Fractional Factorial – Algorithms A and B.....	78
E.2. Fractional Factorial – Algorithm D.....	78
E.3. Response Surface – Algorithms A and B.....	79
E.4. Response Surface – Algorithm D.....	80
E.5. Confirmation Experiment.....	81
E.6. Comparison Experiments.....	82

Word count of main body of report: 13140

Figures

i. Expected Value formula for Boltzmann Selection.....	14
ii. Example of crossover.....	16
iii. Heuristic Crossover Function.....	17
iv. Demonstration of a crossover “mutation”.....	18
v. Fitness function used in timetabling algorithms.....	31
vi. Screen grab of GUI.....	33
vii. Fitness function repeated for convenience.....	45
viii. Summing the local fitnesses of all modules.....	45
ix. Graph of Generations to Reach a Feasible Solution.....	59
x. Graph of Fitness Over Time.....	59

Tables

A. Fitness Function Calls.....	46
B. Factors for the Fractional Factorial Screening Experiment.....	48
C. Results of 26^{-2} Fractional Factorial Experiment for Algorithm A.....	52
D. Results of 26^{-2} Fractional Factorial Experiment for Algorithm B.....	52
E. Results of 27^{-2} Fractional Factorial Experiment for Algorithm D.....	52
F. Results of Response Surface Experiment for Algorithm A.....	54
G. Results of Response Surface Experiment for Algorithm B.....	54
H. Results of Response Surface Experiment for Algorithm D.....	55
I. Optimal Values Found	56
J. Results of Confirmation Experiment	56

1. Introduction

1.1. Manual timetabling

Timetabling is a part of the large field of scheduling. It can be divided into several different problem categories of which exam timetabling and lecture timetabling are prominent examples. It is classed as a problem of NP-hard complexity, effectively ruling out efficient automation by traditional deterministic algorithms.

Even finding a timetable for a modest number of rooms and classes can be highly complex. At the School of Computing within RGU there are 4 modules per semester per course and 7 undergraduate courses. Additionally, each module is divided into around four sessions (individual lectures, tutorials or labs). This problem is coupled with the inclusion of other events (postgraduate courses, meetings etc.) and a variety of other constraints such as room size and allowance of suitable break times.

Traditionally, timetables have been constructed by hand and then modified as appropriate each year (A process known as local repair). This is a laborious process and it would be desirable to automate it in some way.

In this project an attempt is made to follow the work done [4, 16, 25] on using Genetic Algorithms (GAs) and Memetic Algorithms (MAs) to solve the timetabling problem; the stochastic nature of both types of algorithm gives them potential to perform well in this area. Three different implementations (2 GAs, 1 MA) are created, optimised and compared, allowing an observation to

be made on which of the three candidates is the best approach to the problem.

1.2. Alternative Approaches to the Timetabling Problem

1.2.1. Overview

There are many different approaches to timetabling. Genetic Algorithms have been shown to work well when applied to other scheduling problems [12], and work has already been done [3] on using Genetic Algorithms to solve the timetabling problem. This can be improved by using steady-state GAs [14] and an extension of GAs called memetic algorithms [4, 25]. Solutions have also been demonstrated using tabu search, tiling algorithms, simulated annealing, agents and other algorithms. Often the problem is interpreted as a graph colouring problem [17], although some other approaches have also been taken (such as in [25]). Before moving on to look at genetic and memetic algorithms in detail, it is worth looking at some of these alternatives.

1.2.2. Tabu Search

Tabu search is an algorithm which makes extensive use of local search [20]. As it proceeds through the search space it avoids local minima (the major problem associated with local search algorithms) by modifying the set of neighbours around the currently selected solution. The way it achieves this is by building up a tabu-list of already visited solutions – this can also contain solutions not visited but undesirable in some way – these solutions are

ignored during the search. It has been shown to work well with problems like the timetabling problem [8, 19] but has also been outperformed by genetic algorithms in some studies [18].

1.2.3. Tiling Algorithms

A tiling algorithm collects the classes to be scheduled into clusters known as tiles. Each of these tiles hold classes which can run simultaneously; these tiles are then assigned times using a separate search algorithm of some kind. This approach was used with some degree of success in [10], but only in situations such as that in a high school where several classes of students sit the same subject simultaneously. These groups of classes are clustered into the tiles for scheduling – this does not tend to happen in a university timetable where cohort groups sit far more varied courses.

1.2.4. Simulated Annealing

Simulated annealing [12] is an optimisation technique which simulates the behaviour of metal atoms during the process of annealing (a treatment involving extremes of temperature). A temperature is set which reduces over time; this temperature is used to determine the maximum size of random leaps it makes within the search space (mutating candidate timetables by varying amounts). It has been used in conjunction with GAs for the timetabling problem, demonstrated in [11].

1.2.5. Agents

Multi Agent Systems, such as that described in [9], employ several software agents communicating with each other working towards different goals. Each agent can be set up to view the timetable from a different perspective and amends it until a stable timetable satisfying all agents is found.

1.2.6. Why Choose GAs Over the Other Approaches?

Based on the literature study, the three approaches examined in this study will be two variants of a genetic algorithm and a memetic algorithm. This is largely because of previous experience in Genetic Algorithms and because of the relative similarity of the three approaches. This will allow recycling of code and reduce the implementation phase of the project, allowing more time for the optimisation and comparison stages.

2. Background Theory

2.1. Genetic Algorithms

Before proceeding to the practical details of implementation it is appropriate to look at the theory of genetic and memetic algorithms in some detail.

2.1.1. History and General Principals

Genetic algorithms (GAs) are a specialisation of evolution programs, based on the principals of natural selection and random mutation from Darwinistic biological evolution. They were formalised in 1975 by John Holland at the University of Michigan and have been growing in popularity since, particularly for solving problems with a large irregular search space of possible solutions [13]. The basic concept of a GA is that a population of individuals is maintained; each of these holds a chromosome which encodes a possible solution to the problem being solved. With the passing of time the members of the population interact and their content is passed on through generations of new individuals. Fitter solutions (those closer to the optimum) are more likely than their poorer counterparts to “breed”, passing on parts of their genetic material (parts of their solution to the problem) to the individuals (“offspring”) in the next generation.

The first GAs all used a binary encoding scheme – chromosomes were simply strings of 0s and 1s. To illustrate, say a solution consists of a set of numeric parameters (range 0-15) to be entered into some engineering process. To

encode the sequence of values (2,5,12,7,1), each value would be converted to its binary equivalent and the set of values simply concatenated together to form 0010010111000001. (Each value in this string is known as an allele) This was the method preferred by John Holland (their inventor) for numerous reasons [13]. Alternative encodings have since been shown to offer comparable if not better performance in some situations [2, 7, 13] – GAs now exist where each chromosome is a string of bits, integer or real numbers or other values. Far more complex structures such as trees have also been shown to work well in certain situations [13].

There are three major operations involved in evolving the population of a typical GA. In no particular order, these are selection, crossover and mutation.

2.1.2. Selection, Elitism and Steady State GAs

The method by which individuals are chosen to contribute material to the next generation is known as selection. The aim is to give preference to individuals of a higher fitness in the hope that they pass the elements which make them better on to the next generation. This must be carefully balanced so as not to allow suboptimal highly fit individuals to take over the population and wipe out any useful information that may be held by those of a poorer fitness (The balance between these two goals is called the selection pressure). The initial approach to this was a simple probability based system where the likelihood of an individual reproducing directly corresponded to its fitness relative to the rest of the population. This is known as Roulette Wheel Selection because its

operation is similar to that of the selection of numbers on a roulette wheel. This seems like a logical approach to take although in practice often performs poorly because it yields a high selection pressure. This means it tends to allow suboptimal fitter chromosomes take over the population before the high fitness components of less fit individuals are allowed to spread. Attempts to improve basic roulette wheel selection include the use of linear scaling (raw fitness values are replaced with their relative rank) and stochastic universal selection (which reduces the unpredictability of the number of times an individual is selected). Both of these are covered in depth in [13].

Several other approaches to selection have been taken – one of the more commonly used is Tournament Selection [12]. Here, two individuals are selected at random and placed into a “tournament”. Simply, one of the two is chosen randomly with that having the higher fitness is given a higher likelihood of being chosen. This has a lower selection pressure [13] allowing the population to maintain a good diversity. It also requires less computing power than most other methods by only using three random number generations and one comparison operation.

One other approach is Boltzmann Selection [13]. This keeps the selection pressure low early on in the evolutionary process, keeping the population varied and giving all individuals a good likelihood of contributing to the final solution. The pressure is then increased over time, encouraging the population to gradually converge to a highly fit solution. To achieve this, each individual is given an “expected value” $\text{ExpVal}(i, t)$ generated from a formula

[13] such as that illustrated in figure (i). The expected value can then be used in place of raw fitness in a scheme such as roulette selection. In this example, $f(i)$ is the fitness of the individual, T is the current temperature used to set selection pressure and $\langle \rangle_t$ is the average over the population at time t . The effect of this formula is that as the temperature T decreases, the difference in expected value between highly fit and poor chromosomes increases (this leads to an increase in selection pressure). The value of T is decreased slowly with the passing of time, possibly as a function of the number of generations passed or the current best fitness found.

$$\boxed{ExpVal(i, t) = \frac{e^{f(i)/T}}{\langle e^{f(i)/T} \rangle_t}}$$

Figure i

This idea can be expanded to reach what is known as a steady-state genetic algorithm [13]. In this situation the population is not replaced each generation – only a few chromosomes are taken out and replaced (normally a selection of the poorest ones). This is equivalent to having a large number of elites and is closer to the overlap of generations found in biological populations.

2.1.3. Crossover / Recombination

It is useful to think of chromosomes as being made up of several component parts, or genes. These are groups of alleles which encode a particular feature of the chromosome; just as a particular gene in an animal could encode skin or eye colour, a gene in a GA represents one aspect of the solution. [13] Crossover is the process by which genes and particular combinations of several genes (known as schemas) from one chromosome can be reassembled with genes from another to generate new offspring chromosomes (Just as parents contribute different parts of their genetic makeup to their children). The hope is that this process may combine a good schema from one average fitness chromosome with a different good schema from another chromosome to produce a new higher fitness chromosome.

The original approach to this was to select a random point [7, 13] in the chromosome and swap the content of the chromosomes thereafter. This would produce two offspring, as illustrated in figure (ii).

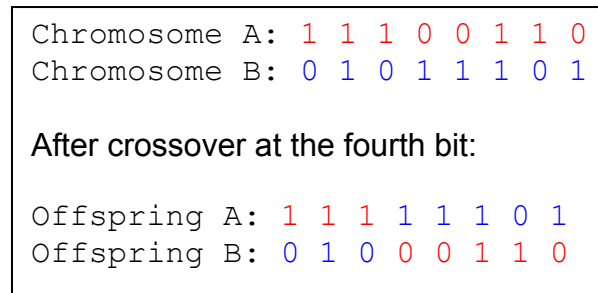


Figure ii

In addition to varying the crossover rate, the number of points at which crossover occurs during the chromosome copying can also be varied. [13] This can offer a large improvement over the one-point crossover previously described. In the example above, one point crossover will never produce a child whose first bit and last bit are either both 0s or both 1s. This would be allowed to happen if there were two points of crossover, where only a portion in the middle of the chromosomes would be swapped between A and B. With two point crossover there is also nothing to stop the points occurring at the same location, effectively returning to 1 point of crossover and allowing the offspring that it makes possible to still occur. Uniform crossover [7, 12] is an extension of this idea which takes each allele in the child from one of its parents at random. This works well in situations where the relative positions of alleles is less important but some researchers remain sceptical because it

ignores any schemas contained in the chromosomes [13]. Uniform Crossover can be adapted to Fitness Based Scan crossover [6] in which the alleles are to be passed on are selected with a probability related to each parent's fitness.

Chromosomes made up of a string of integers or real values can of course still employ the standard crossover although this does not extend to some of the more exotic encodings such as trees. One opportunity which the use of a different encoding yields is the chance to develop entirely different crossover operators in addition to or as a replacement of the standard one. One example is the Average Crossover operator outlined in [7]. This takes an allele from the same position in both parents, and the resulting child allele is the average of these two. This could also be extended to take in more parents, and can also have a weighting assigned to one of the parents when calculating the average (either chosen randomly or based on fitness) [2]. One problem with this approach is that it tends to guide alleles to the midpoint of their range and does not favour extreme or near-boundary values which are often found in optimum solutions. This said, it has also been shown to work well in some limited situations [7].

A further crossover operator is Heuristic Crossover. [6, 12]. This operator uses the fitness function to guide the search direction and differs to the others

[12]

Figure iii

outlined in that it may not result in the successful creation of an offspring. A new chromosome x3 is created using the formula in figure iii:

where r is a random number between 0 and 1, and parent x_2 is not worse than x_1 . Occasionally this will produce an offspring with allele values out of the required range. In this case the process can either be repeated with a new random number or return with no new chromosome generated.

It is because of these alternative versions of crossover that the operator is perhaps better known now as recombination – the process of recombining the components of a chromosome.

The reason that the crossover operation is useful to a GA is still not fully understood. As well as recombining “good” alleles it can also be said to be a macro-mutation operator [13]. The offspring generated by a crossover operation can be vastly different from its parents (for example 00000000 and 11111111 crossed at bits 2 and 6 giving 01111000), resulting in exploration of an entirely different part of the search space. In addition to this, if crossover occurs at a point in the middle of a group of alleles encoding a numeric value a different kind of mutation occurs. This is illustrated by the example given in Figure (iv).

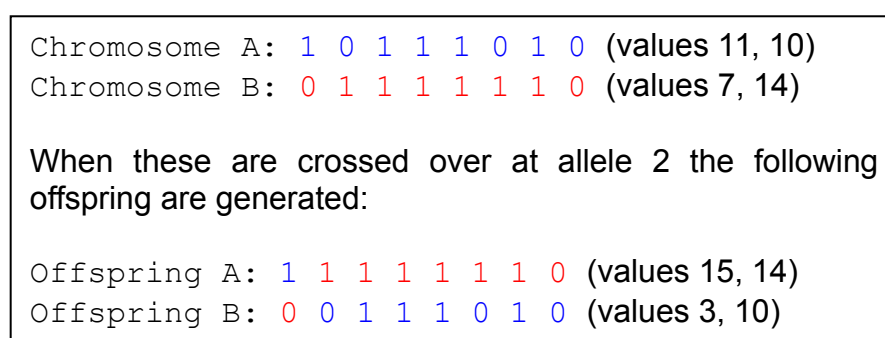


Figure iv

The new values of the first number being represented are much different to what they were. It might have been that the optimal value for this number was 10 and both 7 and 11 were previously close to this; they have now been far removed from it. While it may be desirable to have an additional mutation operator allowing further coverage of the search space, the uncontrollable nature of this mutation may be unwanted. This effect can be avoided by restricting the crossover operation to “safe” boundaries between encoded values or by using an alternative encoding such as integers where groups of alleles are not so closely related.

2.1.4. Mutation

Although crossover creates new individuals in the population, the mutation operation is generally the primary means by which completely new areas of the search space may be explored. During the creation of a new generation of the population there is a small probability that the new offspring will be mutated. Mutation simply involves altering the offspring randomly in some way. Logically, one parameter that may be altered here is the rate at which mutation occurs. Typically this is fairly low, with the probability of a mutation occurring what creating an offspring being around 0.1.

Originally mutation of a chromosome meant that some of its alleles would be randomly flipped from 0 to 1 and vice versa. (How many of the alleles are changed in one mutation operation is another factor which may be varied) Depending on the encoding this could have a large effect on fitness similar to

that of crossover at a poor location, as it is possible that a bit being flipped is the most significant bit of one of the encoded values. Similar to the mutations problem with the crossover operator, it is feasible that a value whose possible range was 0-15 could be mutated from a 2 to a 10 (0010 mutated to 1010). This may well be desirable when looking to expand coverage of the search space; likewise, we may want to be less destructive when altering what may be a reasonably fit chromosome. To control this effect, the operator can be restricted to only altering certain alleles within a chromosome or using Gray encoding (instead of binary, the groups of bits for each number are ordered such that changing a value by one always only results in only one bit changing).

As with crossover, other mutation operators have also been developed. Real and integer value encodings allow mutation of encoded values while still respecting their range. As described in [7], integer values can either be replaced by entirely new random values or can be crept a limited amount from their current value. This could either be a fixed step up or down, a random bounded value up or down, or something more sophisticated such as the use of a convex space function [2].

2.2. Memetic Algorithms

A Memetic Algorithm (MA) is a specialised version of a Genetic Algorithm – although they are a reasonably simple extension of the GA concept they are a reasonably new area of research. Based on the concept of memes rather than genes, it employs a heavy use of local search in addition to the standard

genetic operations. Like genes, memes are passed down through the generations as the evolutionary process runs. The difference lies in the idea that memes can be altered at each generation as they are passed on [4].

In practice this is achieved by the addition of a local search to the normal GA operators. Whenever a new chromosome is created (through mutation or recombination) a local search is performed on it to push it towards a local optimum. While this local search does require some extra processor time, it is hoped that it will reduce the search space of the GA to the subspace of local optima and that this reduction will lead to an overall performance improvement.

3. GAs and MAs Applied to Timetabling

3.1. GAs and Timetabling

One of the most obvious implementations is a simple GA which works directly on candidate timetables, as demonstrated in [16]. Each chromosome would be large, holding an allele for each class to schedule. The GA would assign a room and timeslot to each class (giving each allele a large range of possible values) and its fitness would be a function of the number of constraint violations. This places a heavy onus on the fitness function to guide the search to a working timetable; possible constraint violations would include assignment of classes to undersized rooms or rooms of the wrong type in addition to clashes between classes. This is one of the GAs that studied in this project – for brevity this will be referred to as Algorithm A.

An alternative method would be to have the Genetic Algorithm only assign the timeslots to modules, as used in [25]. This considerably reduces the search space and in turn speeds up the algorithm. Modules can then be assigned to rooms using a greedy algorithm based on room and class size. The specifics of this greedy algorithm will be described in depth later. This method also has the advantage of guaranteeing that modules will only be placed in rooms of the correct type and adequate size; this method of “hard coding” the room size/ type constraint into the algorithm reduces both the complexity of the fitness function and the workload of the GA itself (A large number of infeasible timetables have been removed from the search space). This will lead to an overall improvement in performance if the processing cost of the greedy

algorithm is less than the reduction in processing cost of the GAs. This type is also studied by this project, referred to as Algorithm B (For reasons outlined in the discussions on crossover and mutation, both will use an integer encoding).

An alternative to these approaches is to use the GA to create a permutation of classes to schedule, which is then passed to another algorithm. This algorithm would then assign both timeslots and rooms to classes in the order presented to it by the GA. In the instance the GA would be simply sorting the classes into an ordering which makes them easy to schedule. Given that the focus of this project is the study of a practical implementation of a GA – and that the GA would have only a small role and this “other” algorithm would be doing most of the work – this approach will not be looked at in more detail here.

3.2. MAs and Timetabling

There has been significant research covering the application of MAs to the timetabling problem. One such implementation [25] has been shown to work well on a similar problem to that of the School of Computing, specifically the scheduling classes at Napier University. Memetic Algorithms have also been used to solve the similar problem of exam timetabling [4].

It is quite possible to base an MA on the Algorithm A GA described earlier; in fact work was started on such an adaptation (Algorithm C). This would have added a local search element to the original GA by repairing clashes and room size/type constraint violations as they were found. A study of literature

however leads one to conclude that the simple implementation used in Algorithm A will not perform well and is useful for inclusion because it is the most obvious implementation. This considered, Algorithm C was unlikely to yield a significant improvement and was deemed unnecessary for the project. This led to its abandonment in favour of concentration on the other implementations.

Basing the MA on the second GA described previously (that using the greedy algorithm to assign rooms), we have an algorithm purely responsible for assigning timeslots to classes such that they do not violate the time constraints placed on them (Room assignment being left to the greedy algorithm). The major consideration is how to design the local search.

Local search in [25] is based around the permutation approach for implementation, but the implementation in this project is a graph colouring algorithm and so this cannot be adopted. Based on [4], an alternative possibility is essentially a hillclimbing method. This involves looping through each of the modules, adjusting the timeslot for each and reapplying the greedy algorithm to assign rooms. This can either be a random timeslot reassignment, a slight adjustment or a timeslot chosen so as not to clash with any neighbours (the last of these being that chosen for this implementation). The change in fitness can then be calculated and if there is an overall improvement, the process is repeated. A factor to consider here would how many unsuccessful attempts to improve fitness would be allowed to pass before it could be concluded that there is no further gain to be made. The

adjustment to each timeslot can be a slight adjustment up or down, a completely new random value or a value chosen to not clash with any neighbours. This will require a considerably faster than normal fitness function because the nature of local search with hillclimbing requires a large number of fitness evaluations. An ideal solution would be to calculate the change in fitness which altering a single module's timeslot will cause and add or subtract this from the previously calculated overall fitness as appropriate.

This final implementation is known within the project as Algorithm D. With the omission of C, there are three algorithms to implement and compare. The practical implementation of these will be examined shortly.

4. Practical Implementation

4.1. Practicalities of the Problem

The first stage of the project was a literature study, taking a several weeks. This covered genetic and memetic algorithms and previous work done on automated generation of timetables.

Having studied the literature on past work in the area, several decisions on the design of the algorithms could be made. The aim of the project is to study different implementations of genetic algorithms applied to the timetabling problem. The three algorithms to be implemented are the two GAs and the MA previously described and referred to as algorithms A, B and D.

4.1.1. Constraints

From study of previous work done on timetables [4, 16, 19, 25], analysis of the sample data and consultation with the Roger McDermott (School of Computing timetabler), several possible constraints on any generated timetables have been found. These can be broadly categorised into hard constraints (the breaking of which results in an infeasible timetable) and soft constraints (which do not have to be met, but which lead to desirable timetables when met).

The hard constraints being considered are:

- H1. All classes must be scheduled a room and time
- H2. No clashes (at any one time, a lecturer has 1 class, a student has 1 class, and a room has 1 class in it)
- H3. Room capacities not exceeded
- H4. Correct room types used (lecturers in lecture theatres, labs in laboratories)

The soft constraints are:

- S1. Classes should be scheduled within preferred hours (for example, omitting Wednesday afternoons)
- S2. Distances between classes minimised (keep cohorts in the same building over the course of a day where possible)
- S3. Hour for lunch is allowed between the hours of 12 noon and 2pm
- S4. Bunch classes into groups (don't leave huge gaps) and try not to have a single class in a day
- S5. Try not to have a day or a long run of all lectures

Each of these constraints is given a weight to allow fine tuning of the algorithm, these weights being simply floating point values which reflect the relative importance of the constraint against the others. Constraints H3 and H4 are built in to Algorithms B and D; these both use a greedy algorithm to assign rooms to classes which will either assign them valid rooms or no room at all (resulting in a violation of H1 instead).

The soft constraint S2 was not implemented into the fitness function; this required extra information relating to travelling distance to be incorporated into the room data and this was not available.

The levels used for the weightings are somewhat arbitrary values and may be adjusted if the resulting timetables are undesirable. Currently they are set so that all hard constraints have an equal weighting of 1 and the soft constraints all have an equal weighting of 0.01. Brief experimentation revealed that leaving the constraints of each class (that is, hard or soft) at equal levels yielded good results; further examination of this could be an area for further study.

4.2. Class Structure

NB – UML class diagrams for the Timetable and GA are found in Appendix C.

4.2.1. The Requirements and Timeslots classes

The first stage in developing the algorithms was to build a structure in which the components of the required timetable – the modules to schedule and the cohort groups and lecturers associated with them – could be stored. This Requirements class holds sets (TreeSets, to speed in-order iteration of the objects) of Lecturer, Cohort and Module objects, together with methods to add lecturers and cohorts to modules and iterate over each of the sets. Each of these classes is basically a data wrapper. A module is the timetable application is not a module in the sense of a group of classes; it is a single

session within a module such as an individual lab or lecture. A Module object holds the module number, an identifier to separate it from other parts of the same module, the size and type of room it requires, and the number of timeslots it occupies. The room size required is initialised to zero and is altered as the module is added to cohorts. Modules may be compared either by identifier (the default sort order) or by room size required. The Cohort and Lecturer objects are very similar, so much so that consideration was given to making them both subclasses of a generic Person class, although this was deemed unnecessary. Each object of these classes stores a set (again a TreeSet) of modules with which the lecturer or cohort is associated, as well as an identifier (cohort or lecturer name). The modules are actually wrapped within the node objects from the Timetable class – these hold the time and room assigned to that module making it a trivial task to construct a timetable for an individual person given only the Cohort or Lecturer object (the use of nodes rather than Module objects was a late improvement, discussed later). They also have methods for retrieving this data and for comparing with each other alphabetically by identifier. Cohorts additionally store the number of students they comprise of and when adding a module to a cohort this size is also added to the room size required by that module.

The timeslots available to the timetable are stored in a dedicated Timeslots class which also holds the rooms available to schedule modules into. This class provides methods for iterating over the rooms available, determining which timeslots are available and for changing the availability status of rooms and timeslots. The reasoning for placing the timeslots in a separate class is to

allow the reservation of particular rooms/timeslots. A possible situation in which this would be required is when the school shares its rooms with another and the other school has already reserved some of the rooms at specific times. Due to time constraints the examination of this area of the timetabling problem had to be omitted; the option to allow it had to be built in to the project from an early stage so was added before it became unnecessary.

4.2.2. The Timetable class

The next stage was to create a structure in which completed timetables could be stored and evaluated – the Timetable class. As the problem is in essence a graph colouring problem, a graph data structure is used to hold the timetable. Each node in this graph represents a module to be scheduled, together with the timeslot in the week that has been assigned to it and the room number. Edges link together nodes (modules) which cannot occur simultaneously; examples would be classes with a common lecturer or cohort. The timeslot would be considered to be the node's colour; thus neighbouring modules running at the same time would indicate a clash.

In addition to methods for data access and assigning timeslots and rooms to classes this class also has a method to compute the fitness of a given timetable. It achieves this by totalling the number of violations of each constraint, then multiplying these by their preset and adding them together. Originally this value was then subtracted from 0, giving a fitness ranging from a large negative number (many constraint violations) up to 0. This yields a

high selection pressure and accordingly was found to give a poor performance. Following discussion with the project supervisor, this was replaced with the formula illustrated in figure (v). Here, v is the total number of weighted constraint violations – this function yields fitnesses from 0 to 1. Given the weights discussed previously, v is calculated by adding the total number of hard constraint violations to 0.01 multiplies by the number of soft constraint violations.

$$\text{fitness} = \frac{1}{v}$$

Counting the number of violations of each constraint was delegated to a number of helper methods. Counting the number of classes not assigned to a timeslot or room is a trivial task. (This includes invalid assignments such as room size exceeded or incorrect type, though these only occur with chromosome type A, as the greedy algorithm in B and D guarantee a module being assigned a valid room or none at all) It is achieved by simply iterating over all the nodes in the Timetable and counting those with null values for timeslots or rooms. Counting the number of clashes is also reasonably straightforward, and best demonstrated by the following algorithm:

1. For each node (class) in the timetable, repeat:
 - 1.1. For each of the current node's neighbours found after the current node, repeat:
 - 1.1.1. If the neighbour has been assigned a timeslot that causes it to overlap with the current node (taking the starting timeslot and the length of both into account), increment the clash count by 1.

Step 1.1 needs a little explanation; if all of each node's neighbours were to be considered, we would count each clash twice. The nodes are all given index

numbers so we can step through them in the same order each time – to avoid the doubling up effect we simply look at neighbours which occur after the current node and not those before.

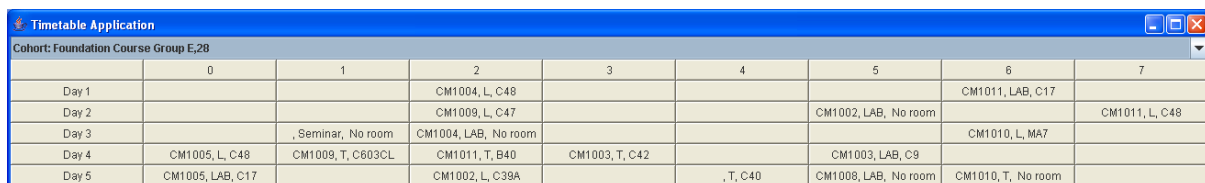
Originally a clash was detected if classes were scheduled to the same timeslot. However, this was invalidated once the ability for a class to be longer than one hour (one timeslot) was added. Now the timeslot assigned to a class is the time at which it starts, it then has another number to determine its length. A clash has occurred if neither of the following conditions is true:

- The current module finishes before the neighbouring module starts
- The current module starts after the neighbouring module ends

Counting the violations of soft constraints is considerably harder because they evaluate the timetable from the perspective of cohorts and lecturers rather than from that of the modules. Each lecturer and cohort is taken in turn, and the timeslots and rooms allocated to each class they are associated with are used to assemble their personal timetable. This may then be parsed to ensure that modules are grouped well and occur at desirable times and sites. Any violations are added to the totals.

4.2.3. *TimetableGUI class*

A simple class to display the timetable in a more human-readable fashion was required to demonstrate that the created solutions were viable timetables. This class creates a grid showing a timetable for a week with a drop-down list of all cohort groups and lecturers. Whenever the drop-down list is changed, the Timetable object is parsed for all modules related to that particular cohort or lecturer. This list is then used to build a standard weekly timetable grid. The GUI is not extensively used in the project, merely serving as a debugging tool more than anything else. A sample timetable display is given in figure (vi).



	0	1	2	3	4	5	6	7
Day 1			CM1004, L, C48				CM1011, LAB, C17	
Day 2			CM1009, L, C47			CM1002, LAB, No room		CM1011, L, C48
Day 3		, Seminar, No room	CM1004, LAB, No room				CM1010, L, MA7	
Day 4	CM1005, L, C48	CM1009, T, C603CL	CM1011, T, B40	CM1003, T, C42		CM1003, LAB, C9		
Day 5	CM1005, LAB, C17		CM1002, L, C39A		, T, C40	CM1008, LAB, No room	CM1010, T, No room	

Figure vi

The GA and MA are well suited to an object-oriented implementation. Each individual in the population is represented as a separate object of a Chromosome class. The generic Chromosome class is largely abstract, simple requiring that each chromosome has methods for evaluating its fitness, mutating itself and crossing itself with another chromosome of the same type. Extending from this foundation, there is the IntegerChromosome class which has alleles represented in an array of integers and methods for mutation and crossover of integer array values, with only the fitness calculation omitted. This is then extended by the TTChromosomeA, TTChromosomeB and

TTChromosomeD classes, corresponding to the algorithms A,B and D being investigated. Each defines the number and range of the alleles to be appropriate to the particular algorithm being used (the number of alleles is the number of modules to schedule, the range is large for type A and smaller for types B and D as they only have to assign timeslots and not rooms as well).

Each of these chromosomes also defines a fitness function. Here, the particular algorithm being used takes the allele values and uses them to assign timeslots and rooms to the modules contained in a Timetable object. The fitness function of the Timetable object is then called to compute the Chromosome's fitness. The TTChromosomeD class also extends the crossover and mutation methods to add the local search immediately after those operations have been performed. Additionally, to reduce computation cost the fitness function for each chromosome is only run the first time it is called. After this the fitness values is stored in a variable and this is returned as necessary. There is a Population class which holds the current generation of chromosomes in an array. This holds methods for selection (which are the same regardless of chromosome type), and makes use of the fitness, crossover and mutation methods of the chromosomes to evolve the population. The core parts of the genetic algorithm outlined here were also used in a previous work [2] in a different area of GA research, and were incorporated into this work in a self-contained Java package. One of the aims of this project is to build on and extend the knowledge gained during that work; this is achieved in one way by building on work already done.

4.2.5. *GA Operators*

Initially the selection operator used was tournament selection. Having been previously used in an alternative setting, the GA package also contains methods for performing roulette wheel selection, linear roulette wheel selection and stochastic universal sampling (described earlier and covered in detail in [13]). These did not perform well in initial tests and following the aim of the project to expand on previously gained knowledge of GAs, a new alternative was chosen for investigation. This is the previously described Boltzmann Selection, an attempt to vary selection pressure over course of the evolutionary process.

When specifically mentioned in papers found during the literature investigation, previous implementations used variants of the standard crossover rather than arithmetic methods. Experimentation also showed arithmetic crossover operator such as Average Crossover perform poorly in timetable generation. This makes some sense – the assignment of modules to specific timeslots is an ordering rather than a numeric problem. Thus timeslots near to each other may have completely differing impacts on fitness. For example, even if a module clashes with nothing at 9am and 1pm on a Monday that does not mean that it will be free of clashes at 10am, 11am or 2pm. Thus averaging the “good” values of 9am and 1pm together to give 11am will not necessarily result in a fitness improvement (in fact it is possible that the “good” values are overwritten by poorer ones). It follows that the traditional crossover would perform well. In the above example assigning a value of either 9am or

1pm to a module will make a positive contribution to the chromosome's fitness; both of the offspring generated will have one of these preferable values. Given this, the plain crossover operator is used in the implementations being tested.

The problems with numeric crossover also affect numeric mutation operators, so one might think it would be advantageous to use a mutation operator which yields random changes rather than anything more sophisticated based on a mathematical formula (which would also require more processing time). That said, it could be said that some timetables only need "tweaked" slightly to become feasible (that is, no hard constraint violations). This could involve making only slight mutations rather than large jumps – for example, changing a start time from 2pm to 3pm. Later in the evolution process this would also have the potential to improve timetables by shifting modules from long runs of classes or from occupying the lunchtime period. Clearly this is an area of uncertainty that needs further investigation. This is achieved by using the Creep Mutation operator [7], in which mutations can alter an allele by a random value up to a constant creep step. During the optimisation process for the algorithms this creep step will be one of the factors, allowing the best value (low values resembling a gentle creep, high values resulting in more random jumps) to be determined. It is also feasible that this creep value should decrease over time to allow large jumps early on during evolution and smaller jumps during the final tweaking of the timetable when soft constraints become more important. Variation of the creep step will not be considered here due to time constraints but is another possible area for future study.

4.2.6. Greedy algorithm

Algorithms B and D employ the use of a greedy algorithm to allocate modules to rooms once they have been assigned timeslots. This is a relatively simple algorithm to implement.

Initially, the set of modules for one particular timeslot were sorted into descending size order (then ordered by room type). The algorithm would then take each room and proceed through the list of rooms in decreasing size order. This way, the modules needing larger rooms (the harder ones to allocate) would be given rooms first. Unfortunately this ordering results in small classes being assigned to rooms larger than they require (potentially a tutorial group of 15 students could be placed in a 200 capacity lecture theatre) – not a bad problem but one it would be desirable to avoid. Fortunately this is easily solved by reversing the sorts. Modules and rooms are now ordered in ascending size order; if a room is too small it is passed over and a larger one searched for. This way, modules will be assigned to rooms just large enough to hold them.

4.3. Reading the data

The requirements for the timetable (the modules to be scheduled, the rooms to fill and so on) are taken from the timetabling requirements of the first semester of 2000-2001 in the School of Computing. The School uses the Celcat [22] timetabling software to hold its manually created timetables. Following discussions with the school timetabler a number of data files related

to this and other semesters was obtained. The semester chosen was purely because the data for that semester appeared the most consistent and well-structured.

Reading the room, cohort, module and lecturer data was straightforward text parsing. As the data files are read, a new object of the appropriate type is created and added to an array. This array is then fed into the Requirements class.

In contrast to this, adding the links between cohorts, lecturers and modules (and hence defining which nodes on the graph are neighbours) is a little more difficult. The links between lecturers and modules are held in a flat text file; as each link is read, a linear search is performed on the lecturer array to find one with a matching name. Then a linear search is performed on the set of nodes in the Timetable object; the node holding a matching module to that required is then copied into the lecturer object.

A similar process is used to build the links between the cohorts except that there is not a straight flat file in the data provided. Instead, the complete timetable file had to be parsed to find which modules were associated with which cohorts.

The Timetable object created can be reused by resetting all the nodes to undefined timeslots and no rooms. This allows the timetable to be reused without reloading the data. This means that although the linear search and

string comparisons used here are not very efficient or fast, the process is only performed once at program initialisation and thus is not of much bearing on the overall algorithm speed.

Once the code for reading the timetable requirements had been written, the process of running and improving the algorithms could begin. During this, it became clear from an early stage that the algorithms were taking a long time to run. In addition to the improvements discussed shortly, a simpler subset of requirements was created to allow faster tests to be run without the burden of building timetables for the entire school. This subset consists of the modules and cohorts in the undergraduate foundation year – omitting everything for years 2-4 and the postgraduate courses and reducing the number of modules to schedule to around a sixth of that in the full problem.

4.4. Problems with Algorithm Speed

4.4.1. Overview

Once the GAs were implemented they were running but considerably slowly, and generally struggling to reach an optimal timetable. This was the case for all three algorithms so it was likely to be the fitness function at fault (the GA code had already been tested successfully with other fitness functions). The lack of a functioning local search also considerably hampered the performance of the MA.

Several areas were investigated for improvement; during this process the GA/MA would be run while outputting the best fitness found at each generation. This process would be repeated for a few runs to reduce any random anomalies. In conjunction with the GUI, running the algorithms in this way allowed the best operators and likely best ranges for other parameters to be determined.

Initially some experimentation with the basic GA operators was performed. It was at this stage that the arithmetic operators such as Average Crossover described earlier were confirmed to perform poorly. Mutation appeared to make less difference and it was decided to use the most configurable mutation operator (number creep) and allow the optimisation process to improve it later.

4.4.2. Sorting the Modules

In [25] the modules are sorted into order by size of room required prior to commencing the MA. In this case it is a requirement of the permutation based fitness function but it opened another line of investigation. If the modules were sorted somehow, would that allow groups of alleles (genes) matching groups of similarly difficult to schedule modules to form in the chromosomes within the population? After some experimentation with the set of foundation year modules, this appeared to have a positive effect on performance; it approximately halved the number of clashing modules scheduled for the same time in the timetable for the same number of generations. Two approaches were tried; ordering by room size required and ordering by the number of

neighbours (how hard it was to find a timeslot without clashes) – the latter improved performance best. This effect was also reflected in run for all three algorithms.

4.4.3. Boltzmann Selection

A reason for poor performance in many GAs is poor coverage of the search space. This can occur for several reasons but it generally results in a large number of suboptimal chromosomes taking over the population. Two attempts to stop this were made during this investigation. Firstly, a new selection operator, Boltzmann Selection (outlined earlier) was implemented. This reduces selection pressure early on in the evolutionary process allowing a widely varied population and increases it as the search begins to focus on an optimum. Another means of achieving a similar goal was also tried; varying mutation rate. Early on, the mutation rate was kept high (close to 1.0) to allow a highly diverse population to develop (Elitism ensures that the high mutation rate does not destroy the best chromosomes found so far). As the population begins to converge on an optimal solution the mutation rate is lowered. Disappointingly, neither of these approaches yielded a massive gain in performance; varying the mutation rate actually appeared to make some runs poorer. Boltzmann Selection did however show a small positive effect and was subsequently included as one of the selection operators investigated in the optimisation experiments.

4.4.4. The Global Fitness Function

Following investigation into the factors affecting the MA and GA, it was clear that the major place for improvement was the fitness function. From the number of constraints and complexity of timetables described earlier, it can be deduced that the assessment of a candidate timetable is likely to be a lengthy process. Originally the fitness function calculated the number of violations of each constraint separately, adding the results together at the end. This was a logical approach and made the initial implementation straightforward. It did however mean three separate traversals of the graph of modules to count the hard constraint violations; once to check for timeslot clashes with neighbouring modules, once to check for non-allocation of and invalid timeslots and once to check for non-allocation of and invalid rooms. This was an obvious choice for improvement; now only one pass of the modules graph is made, checking for all hard constraint clashes on the way.

Another wasteful loop was found in the calculation of soft constraint violations, reasonably late on in the course of the project. Originally the set of lecturers and cohorts would be traversed and for each a weekly timetable would be constructed. This would involve looking at the lecturer / cohort's modules and finding the nodes holding these modules in the timetable object. The node could then be examined to determine what timeslot and room had been assigned to the module and this data used to build a timetable grid. This process involved a costly linear search for every lecturer and cohort which was likely to be a significant drag on performance. Using the object-oriented nature of Java made fixing this problem easy; rather than storing a reference

to Module objects in each Cohort or Lecturer object a reference to the Timetable.Node object was stored instead. Now the need for the linear search was gone – to construct a timetable for a lecturer or cohort all that must be done is a simple traversal of the relatively small set of Timetable.Node objects held within that lecturer or cohort object. The work of matching modules to nodes is now done during the one-time-only data loading process at program initialisation.

An attempt was made to reduce the number of fitness function executions by storing all chromosomes evaluated so far and parsing this when a new chromosome was created. This would remove any repeated running of the fitness function on identical timetables. In practice, the number of timetables evaluated runs in to many thousands and the set being stored rapidly exceeded the memory of the host computer. Additionally, because the search was linear (sorting the stored chromosomes to allow binary search being even more time consuming) it quickly became slower than just evaluating the fitness function.

4.4.5. The Local Fitness Function

The strength of the memetic algorithm lies in its ability to reduce the total search space by using local search to reach a local optimum whenever a new chromosome is generated. Initially the full fitness function was called each time a new chromosome was generated in the local search, but for the MA to yield an improvement over the GA the local search must take very little processing power. This is achieved by having a “local fitness” function which

can determine the change in overall chromosome fitness yielded by altering one allele (that is, changing one module's timeslot) without recalculating the fitness for the entire timetable. This may seem a straightforward thing to implement but is more complicated than it first appears. Any single change to the timeslot of a module has an effect on all of the following:

1. The number of clashes the changed module has with its neighbours
2. The number of clashes each of those neighbours has
3. The rooms allocated at both the timeslot it was in and the new timeslot, and whether this has a reflection on the number of modules not allocated to rooms

The makeup of the timetables for the lecturer and all cohorts associated with that module

(1) is easy to recalculate and can be done by simply comparing the set of the changed module's neighbours with it; (2) is also simple to implement as an extension to (1), although more processor intensive. (3) requires the greedy algorithm to be called to reassign rooms to modules in the timeslots. This requires a costly linear search of the timetable to find all modules allocated to either timeslot, as well as running the greedy algorithm twice (which includes a sort into room size order). Although (4) had been improved considerably by removing the linear search for modules associated with lecturers / cohorts, this did not improve the local search. It required a linear search through all lecturers and cohorts to find those which are associated with the changed

module before analysing each of their timetables. Fortunately the object oriented nature of the program made improving this straightforward – now each Module object stores a set of references to all Cohort and Lecturer objects associated with it, this being updated when adding Module objects to Cohorts and Lecturers. This further removal of a linear search improved matters somewhat.

Although much work was done trying to improve the local search algorithm, it still runs disappointingly slowly. Perhaps further optimisations are possible – this would certainly be one focus of any further work. One alternative could be to remove the costly calculations relating to soft constraints until much later in the evolutionary process, allowing a useable timetable to be created then making it more desirable. To compound the speed problem, when compared to the full fitness function it did not seem to compute fitness changes correctly. One reason for this is the function used to compute fitness from the number of constraint violations; that previously given in figure (v) and repeated for convenience here in figure (vii).

$$fitness = \frac{1}{1 + v}$$

Figure viii

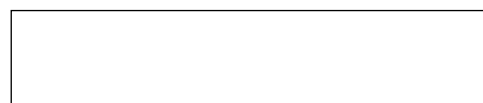


Figure viii

Summing the violations across the whole timetable to find v and substituting into the formula does not yield the same result as summing the violations caused by one module to give u , substituting into the formula for each module and adding all the results together afterward, as in Figure (viii). For example,

say a timetable had 3 modules, the allocation for each of which caused 3 constraint violations. Overall, there are 9 violations, thus we have a fitness of 1 over (1+9), giving 0.1. However, the local fitness impact of each would be calculated as 1 over (1+3), giving 0.3333. Added together, the total “fitness” would be 1.0; clearly incorrect. To solve this, a different formula would need to be used when calculating local fitness.

To ensure fair comparison of the algorithms the decision was taken to use the full fitness algorithm for local searches – although this would be much slower it was at least accurate and as long as the number of local searches performed was counted it would be possible to see how much of an improvement a local fitness function would make. A set of static variables were added to the Timetable class to keep track of the number of fitness functions called from both the global timetable fitness and local search functions. The total numbers of each type of fitness evaluation completed during 5 sample runs to 2000 generations are given in Table A.

<i>Table A – Fitness Function Calls</i>		
	Full fitness function calls	Local fitness function calls
Average	325799.6	5774435
Std dev	559.7569	67285.11

It can be seen that – as expected – local fitness calculations outnumber full fitness calculations considerably; just under 20 to 1 in this case. So here the local fitness function would need to be at least 20 times faster than the full fitness function to yield an improvement. This is not an unreasonable target given that it would only have to evaluate a small portion of the timetable.

5. Optimisation of the Algorithms

5.1. Overview

Prior to comparison of the three algorithms it was desirable that each be performing optimally to allow them to compete on a level playing field. The problem is that optimisation of several interacting factors simultaneously is in itself a computationally hard problem (One that is in fact well suited to a GA solution; indeed much work has been done on optimising GAs with other GAs). However, construction of a further GA to optimise the algorithms would require much more work, beyond the scope of this project.

5.2. Fractional Factorial Screening Experiment

5.2.1. Fractional Factorial Analysis

A full factorial experiment which could establish all interactions between the factors would be ideal, but as the name implies would require 2^n experiments (where n is the number of factors) with each factor having 2 possible values. This rapidly becomes a large number, 26 being 64 and 27 being 128. Fractional factorial analysis is an industry standard approach to optimisation of factors such as those affecting a genetic or memetic algorithm, demonstrated in [15]. It trades off analysis of the higher order interactions to reduce the total number of experiments. Here, fractional factorial analysis will be used for a screening experiment where insignificant factors are determined and removed from further analysis. A response surface modelling of the

significant factors will then be performed to determine their optimal values. The statistical package Minitab [23] provides a good set of tools for optimising multiple factors using this technique. Given a set of parameters to examine with their ranges it will generate a set of experiments to be performed. The results of these experiments are then used to determine the significance of the factors involved allowing the insignificant ones to be screened out from the later response surface experiment.

5.2.2. Factors

The factors for all three algorithms are given in Table B.

Factor	Minimum Value	Maximum Value
Population Size	100	500
Mutation Rate	0.02	0.2
Crossover Rate	0.25	0.75
Crossover Points	2	20
Mutation Creep Step	1	10
Selection Method	Tournament Selection	Boltzmann Selection
Local Search Iterations*	1	10

*Only included for the memetic algorithm (Algorithm D)

The high and low values are chosen based on previous experience with GAs and values used in other implementations. In more detail the factors are:

- Population size is simply the number of chromosomes present in each generation.
- Mutation and Crossover Rate are the probabilities that either mutation or crossover will occur at any one breeding. The separate probability

that one particular allele may be mutated during a mutation operation is fixed at 0.1.

- Crossover Points is the number of points at which chromosomes cross during a crossover operation.
- Mutation Creep Step is the maximum change that may be applied to an allele during a mutation operation
- Selection Method is the technique used to select prospective parent chromosomes. Being a non-numeric factor it cannot be optimised in the strictest sense, but its significance can still be calculated and manual analysis of the results may indicate which method yields better performance.
- Local Search Iterations is the number of repeated unsuccessful attempts at improving a chromosome's fitness that the memetic algorithm local search has before assuming that the local optimum has been reached.

There are also a large number of factors which could also be altered, but will be kept fixed to keep the number of experiments at a reasonable level. The number of elites retained between generations is kept at five. The mutation and crossover operators could also be changed to one of the alternatives discussed earlier but are kept fixed to creep mutation and standard crossover. By altering the maximum creep step, we can have either gentle creep mutation (low creep step) or effectively random value mutation (high creep step), so by including creep step in the optimisation we are effectively looking at two mutation operators anyway. For crossover, the alternatives for integer

alleles include averaging or some other mathematical function of both parents chromosomes and more complex versions of plain crossover. It was felt that mathematical crossover like averaging makes little sense when considering that two timeslots near to each other in the timetable could be populated completely differently (and hence be far more/less suitable for a module). Effectively a mathematical crossover would be a complex mutation operator meaning that an allele in an offspring chromosome should be taken unaltered from one of its parents, not a mathematical function of both.

5.2.3. Approach Taken

The aim of this optimisation is to reduce the overall time that each algorithm takes to reach a viable timetable. This is repeated 10 times each to reduce the effects of randomness.

Although a single program run is considerably faster than the manual timetabling procedure it still takes hours rather than minutes to complete. While it would be most desirable to optimise each algorithm based on the number of generations required to reach a feasible solution (the approach taken in [2]), given the number of repeats required and the large number of experiments required by the fractional factorial analysis (though still far less than full factorial) the decision was taken to look at the average fitness level of the best timetable found after a fixed number of generations. While not perfect, the best fitness found does increase over a reasonably smooth curve for most GAs, so this approach is acceptable. With more time (or had the fitness function itself been improved in its efficiency) it would be better to

repeat the optimisation running each algorithm to completion. The limit chosen was 200 generations.

Another approach that was considered was to optimise each algorithm using the much simpler problem of scheduling foundation year modules only, having less than 1/5 the number of items to schedule. This approach was disregarded on the grounds that the algorithms should be optimised when running on the harder problem which potentially has a much different search space.

Seeded random number generators are used for all random elements of the GA and MA runs. This guarantees that each experiment starts with the same population. The number seed starts at 1000 and is incremented by 1000 for each repeat before being reset to 1000 for the next experiment. The maximum fitness found at each generation in each experiment is output to a text file (guaranteed to be the best fitness in the 200 generation population because of the use of elitism). Although only the best fitness found after 200 generations is required it is helpful for debugging purposes to output as much data as possible and discard that which is not needed later rather than having to rerun the experiments again if more data is required later.

5.2.4. Results

The results of the fractional factorial experiments are given in Tables C-E. The significant factors are those with a p-value less than 0.05 and are shown in bold in the tables. (Detailed data is given in Appendix E)

Table C – Results of 2⁶⁻² Fractional Factorial Experiment for Algorithm A

Factor	Effect	Coefficient	Standard Coefficient	t-ratio	p-value
Constant		0.002406	0.000064	37.79	0
Population Size	0.000594	0.000297	0.000065	4.54	0.001
Mutation Rate	-5.1E-05	-2.6E-05	0.000052	-0.49	0.635
Crossover Rate	-0.00023	-0.00011	0.000077	-1.48	0.17
Crossover Points	0.000227	0.000114	0.000077	1.48	0.17
Mutation Creep Step	-0.00005	-2.5E-05	0.000054	-0.46	0.654
Selection Method	-0.00058	-0.00029	0.00006	-4.87	0.001

Table D – Results of 2⁶⁻² Fractional Factorial Experiment for Algorithm B

Factor	Effect	Coefficient	Standard Coefficient	t-ratio	p-value
Constant		0.001699	0.000061	27.72	0
Population Size	0.000167	0.000084	0.000063	1.33	0.213
Mutation Rate	-0.00014	-7.1E-05	0.00005	-1.4	0.192
Crossover Rate	-0.00026	-0.00013	0.000074	-1.77	0.107
Crossover Points	-6.5E-05	-3.3E-05	0.000074	-0.44	0.667
Mutation Creep Step	-0.00019	-9.5E-05	0.000052	-1.82	0.099
Selection Method	-0.00036	-0.00018	0.000057	-3.15	0.01

Table E – Results of 2⁷⁻² Fractional Factorial Experiment for Algorithm D

Factor	Effect	Coefficient	Standard Coefficient	t-ratio	p-value
Constant		0.001825	0.000064	28.53	0.000
Population Size	0.000501	0.000250	0.000070	3.57	0.016
Mutation Rate	-0.000003	-0.000001	0.000045	-0.03	0.976
Crossover Rate	-0.000006	-0.000003	0.000005	-0.61	0.567
Crossover Points	-0.000059	-0.000029	0.000098	-0.30	0.776
Mutation Creep Step	0.000033	0.000017	0.000092	0.18	0.863
Selection Method	-0.000404	-0.000202	0.000043	-4.66	0.005
Local Search Iterations	0.000162	0.000081	0.000101	0.80	0.460

It can be seen that selection method is significant in all three algorithms. Only in Algorithm A is another factor significant – population size.

5.3. Response Surface Experiment

5.3.1. Summary

Once the significant factors have been determined by the fractional factorial experiments it is possible to fix the insignificant factors at some arbitrary value and “zoom in” using a central composite design response surface experiment to determine the optimal values of the important factors. The response surface is defined by a general quadratic equation in the significant variables. Minitab solves the system of equations resulting from the partial derivatives of this equation, coefficients of the general surface are determined and optimal values for each variable are found.

The only issue here is that the response surface experiment can only optimise quantitative (numeric) factors such as crossover and mutation rate, not qualitative ones such as the selection operator used. Manual examination of the results from the fractional factorial experiments indicates that (perhaps surprisingly) tournament selection outperformed Boltzmann selection. This may be an issue with implementation – perhaps the way in which the temperature constant was calculated was not as it could be. This is a further area for possible future study. With this in mind, all experiments after this point were conducted using tournament selection.

The response surface approach is more suited to multiple parameter optimisations – not the case here because only the population size is being optimised. Initially it appeared as if the fractional factorial experiments showed all the factors to be significant, so the response surface was run to optimise all parameters. Maximum and minimum values for parameter were the same as those used in the fractional factorial screening experiment. This later turned out to be a misunderstanding of Minitab’s operation, and the results given previously are the correct ones. The experiments were not wasted however; it is still possible to use their results to determine an optimal value for the population size and some other interesting pieces of information.

5.3.2. Results

Table F – Results of Response Surface Experiment for Algorithm A

Factor	Coefficient	Standard Coefficient	t-ratio	p-value
Constant	0.002288	0.000026	89.431	0.000
Population Size	0.000121	0.000017	7.282	0.000
Mutation Rate	-0.000032	0.000016	-1.954	0.077
Crossover Rate	0.000071	0.000015	4.851	0.001
Crossover Points	0.000039	0.000016	2.354	0.038
Mutation Creep Step	-0.000017	0.000015	-1.127	0.284

Table G – Results of Response Surface Experiment for Algorithm B

Factor	Coefficient	Standard Coefficient	t-ratio	p-value
Constant	0.001695	0.000029	57.725	0.000
Population Size	0.000033	0.000019	1.716	0.114
Mutation Rate	-0.000045	0.000019	-2.416	0.034
Crossover Rate	-0.000005	0.000017	-0.281	0.784
Crossover Points	-0.000014	0.000019	-0.744	0.472
Mutation Creep Step	0.000015	0.000017	0.898	0.388

Table H – Results of Response Surface Experiment for Algorithm D

Factor	Coefficient	Standard Coefficient	t-ratio	p-value
Constant	0.001568	0.000026	60.796	0.000
Population Size	-0.000007	0.000011	-0.626	0.537
Mutation Rate	-0.000044	0.000011	-3.838	0.001
Crossover Rate	-0.000042	0.000011	-3.724	0.001
Crossover Points	0.000005	0.000011	0.412	0.684
Mutation Creep Step	-0.000014	0.000013	-1.068	0.296
Local Search Iterations	0.000021	0.000013	1.543	0.135

As indicated by the fractional factorial screening experiment, the population size was a significant factor in Algorithm A. Interestingly, the crossover rate and the number of crossover points also appear to be significant ($p < 0.05$) in the response surface experiment.

Algorithm B has no significant factors in the response surface experiment. As selection method was the only significant factor found in the screening experiment and is not included here, this is as expected.

Algorithm D is most interesting. While the screening experiment indicated that population size was significant, here mutation and crossover rate are significant and population size is not. It is uncertain what could have caused this.

The optimum values for the population size (the only significant factor other than the selection method) was the maximum of 500 for both algorithms A and D. Likewise, optimal values for other factors were also at their upper / lower

limits. This would suggest that at least the upper limit was too restrictive; further investigation could well reveal a higher optimum population size.

The optimal values found are given in Table I.

	Algorithm A	Algorithm B	Algorithm D
Population Size	500	500	500
Mutation Rate	0.02	0.02	0.02
Crossover Rate	0.75	0.75	0.25
Crossover Points	20	20	2
Mutation Creep Step	10	10	10
Local Search Iterations	N/A	N/A	10

5.4. Confirmation Experiment

With the optimal values determined, a short confirmation experiment was run with each of the algorithms to prove that they are now performing better than any of their previous runs. This was also over 200 generations but repeated 10 times each; the results for each algorithm are given in Table J.

Algorithm	Average best fitness found after 200 generations	Std deviation	Best fitness found during previous experiments
A	0.002471	0.000050	0.0026401
B	0.001686	0.000045	0.001892
D	0.001601	0.000052	0.0018568

These values were compared with the results from both the fractional factorial and response surface experiments. For each of the respective algorithms the values were within 12% of the best of the previous results, confirming that the

significant parameters being examined were close to their optimal values (allowing for random error). The slight drop is interesting; the optimal values are the same as those which yielded the best results in previous runs, so the drop must be due to some random error. This does not guarantee that those factors not being examined are optimal; however, examination of the large number of variables remaining is beyond the scope of this project and given past experience of GA design the majority of these are not as likely to be significant as those examined.

6. Comparison of the Algorithms

6.1. Experiments

At this point the final stage of the project has been reached. The algorithms optimised, it is now possible to compare their performance relative to each other. Direct comparison of algorithms A and B is possible by counting the number of fitness evaluations run to reach a particular fitness – the fitness evaluation being by far the most expensive part of the algorithms. The memetic algorithm cannot be evaluated directly in this way because the local fitness function is also used; to allow this to be a measure of performance it is necessary to look closely at the relative calculation costs of the full search and the local search.

The fitness reached will be that at which no hard constraints are violated. Given the weightings and fitness formula discussed earlier (100 for hard constraints and 1 for soft constraints) this will occur when the fitness is more than 0.5. In some extreme circumstances (when there are more than 100 soft constraint violations) it will occur at less than 0.5, but under these conditions the timetable will not be appealing to those using it and would likely need some work anyway. Additionally, owing to time constraints near the end of the project (especially considering the lack of a local fitness function in the memetic algorithm) it became necessary to run the comparisons using the foundation year data only and not the full school timetable. With more time the experiments could be performed with the more time costly full data set.

Each algorithm was run to 2000 generations 10 times. The average best fitness found at each generation (effectively the quality of solution) and the percentage of runs having found a feasible timetable at each generation were recorded and are presented here.

6.2. Results

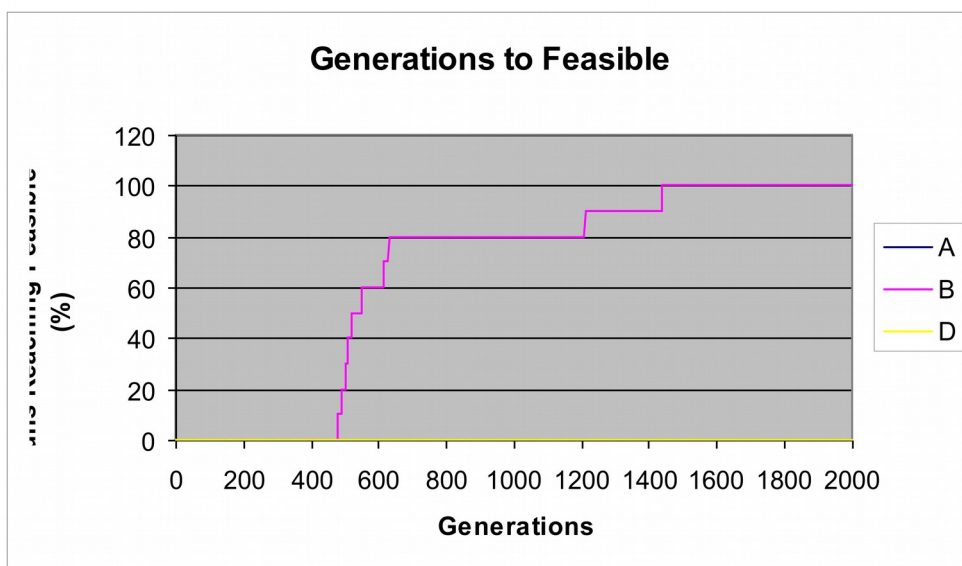


Figure x

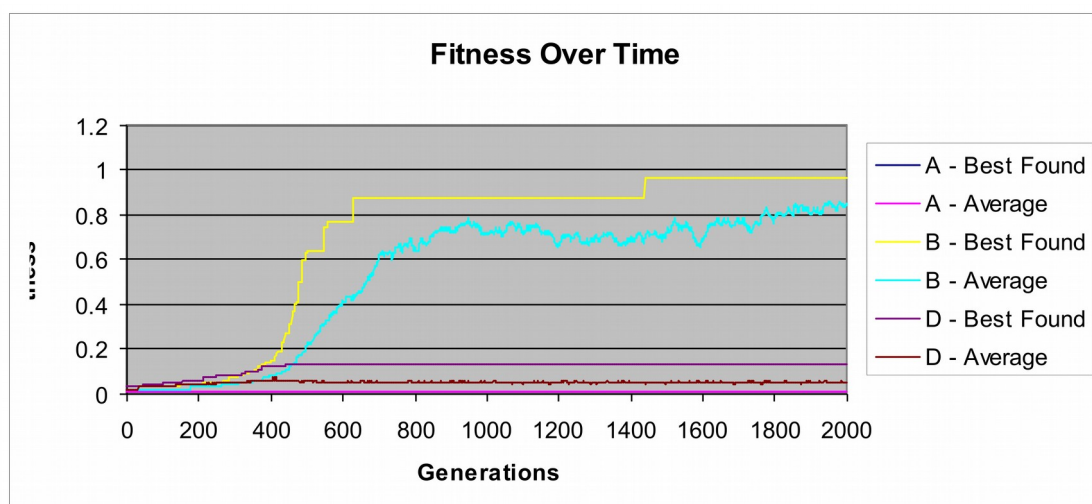


Figure ix

It is clear that with the current implementations, Algorithm B (GA + greedy room allocation) performs the best. Algorithm A performs exceptionally poorly, never raising above very low fitness levels and certainly never achieving a feasible timetable. While it might be expected that it would not perform as well as the other algorithms (as discussed previously) its highly poor performance is interesting. The greatly increased search space brought about by requiring the GA to perform room assignment has obviously made a large difference. One thought here is that with the much larger range of possible allele values, the limit on the maximum mutation is too low. To change the timeslot of a module (and hence repair a clash) it would require a large change to its allele value (on average this would be equal to half the total number of rooms), whereas a change of 1 in Algorithms B and D will change the timeslot. Additionally, it is possible that different operators such as the uniform or fitness based scan crossover would work better. It is also feasible that a new mutation operator which acknowledges the encoding could be developed – this would allow change of either room or timeslot without affecting the other.

The performance of Algorithm B is very good – best illustrated by diagram A, not only does it generate feasible timetables in mostly under 1000 generations, but as indicated by the steepness of its graph it also takes a reasonably constant (500-600) number of generations to do so. In addition to this the average best fitness found is also very high – at the point a feasible timetable is found the fitness is just over 0.5 meaning that there are around 100 soft constraint violations. In around another 1000 generations this improves to a fitness of 0.96 or 4 soft constraint violations. Even across the

smaller foundation year timetable this is quite good – it is not unheard of for students to have what might be considered a soft constraint violation like Wednesday afternoon classes or more than three consecutive hours of lecturing.

The real unexpected performance is of Algorithm D. It was hoped that even though the algorithm ran slower than the others (due to the lack of a working local fitness method) it would still take fewer generations to find a feasible timetable. If this were the case a suitably fast fitness function would result in Algorithm D being faster overall. However, the algorithm appears to stop improving at around 500 generations – just the same point that Algorithm B starts reaching feasible solutions. There are a number of possible reasons for this. It may be that the GA is not able to make large enough changes to the timetables to jump from one local optimum to another and is getting stuck.

6.3. Manually Generated Timetable

An attempt to read the manually generated timetable for the data set being used was also made. This would allow the fitness function to assess the timetable and allow comparison between the results of the time consuming manual process and the automated system. Although this did appear to work, the intricacies of Celcat data format used resulted in several classes remaining unscheduled. This gave the manual timetable a very poor fitness. It is worth noting that the automated system did manage to allocate these classes in addition to the others. It is also a simple task to rerun the algorithm

if an undesirable timetable is generated – this is not the case for the manually generated timetable.

7. Conclusions and Future Work

The project has successfully investigated different applications of genetic algorithms to the timetabling problem. While the original goals of developing the best performing algorithm further and a more advanced GUI were not achieved, the extra time spent optimising the algorithms was a valuable experience. The project did meet the main objective; to implement and optimise three algorithms to solve the timetabling problem and compare them.

The project followed the initial time plan well until the optimisation stage – at this point progress fell behind because of the poor performance of the algorithms. Some extra time was devoted to improving this performance; this was a worthwhile experience and did make an improvement to performance so is not deemed to be wasted.

A hindrance to the progress of the project was the lack of specifics in the research papers about the implementations the work was based on. The resulting program also has extra capacity added for future expansion (such as the ability to change the timetable size and available times using the timeslots class) which was not used. It may have been better to have achieved an efficient working implementation using a fixed structure first and if time had allowed the flexibility could have been added in.

There are several areas which could be expanded on in future research. The literature studied indicates that there is considerable room for improvement in the local search method of the memetic algorithm, especially given the

algorithm's unexpected poor performance. In particular, the local fitness method of the Timetable class needs perfecting to allow the use of it rather than the more costly complete timetable fitness calculation. This may require a reworking of the class structure to allow fitness changes to be more easily determined.

While Boltzmann Selection did not appear to be suited to the timetabling problem, a variety of alternative operators also exist which could be tried. The possibility of varying the creep mutation step over time and use of different mutation and crossover operators as discussed earlier could also be looked into. Candidates include the many different operators looked at in the earlier discussion of GAs.

Some investigation into the weights given to constraints – especially the soft constraints – also remains to be carried out. It would be helpful to conduct a trial in which generated timetables are given to students for evaluation. The closest to this reached in this project was the electronic evaluation of a manually generated timetable; though this did allow some comparisons to be made. The possibility of relaxing soft constraints until a feasible timetable has been developed could have a significant impact on performance and is certainly an area worth exploring.

Further to work on these ideas, completely unexplored areas of GAs applied to timetabling include the permutation implementation detailed in [25] and other implementations not covered here. Timetabling and genetic algorithms

are both vast areas of study and consequently a large amount of research is left to be done.

8. References

8.1. Books / Research Papers

1. Blazewicz, J., Ecker, K.H., Schmidt, G. and Weglarz, J.: *Scheduling in Computer and Manufacturing Systems*. Springer, 1994.
2. Brownlee, A.: *Evolving Better Cancer Chemotherapy with Genetic Algorithms*. Unpublished – Carnegie Funded RGU Summer Project, 2004.
3. Burke, E., Elliman, D and Weare, R.: *A Genetic Algorithm Based University Timetabling System*. East-West Int. Conf. Comp. Techn. Education Crimea 1994, vol. I, pp. 55-40
4. Burke, E., Newall, J, Weare, R. *A Memetic Algorithm for University Timetabling* in Burke, E., Ross. P. (Eds.): *The Practice and Theory of Automated Timetabling I*, pp. 241-250, Springer, 1995.
5. Carmusciano, F., De Luca Cardillo, D.: *A Simulated Annealing with Tabu List Algorithm for the School Timetable Problem* in Burke, E., Ross. P. (Eds.): *The Practice and Theory of Automated Timetabling I*, pp. 241-250, Springer, 1995.
6. DeJong, K., Fogel, L., Schefel, H., et al: *The Handbook of Evolutionary Computation*. IOP Publishing Ltd. And Oxford University Press, (1997).
7. Davis, L., et al.: *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, (1991).
8. Desef, T., Bortfeldt, A. and Gehring, H.: *A Tabu Search Algorithm for Solving the Timetabling-Problem for German Primary Schools* in Burke, E., Causmaecker, P. (Eds.): *The Practice and Theory of Automated Timetabling IV*. Springer, 2002.
9. Kaplansky, E., Meisels, A.: *Negotiation among Scheduling Agents for Distributed Timetabling* in Burke, E., and Trick, M. (Eds.): *The Practice and Theory of Automated Timetabling V*. Springer, 2004.
10. Kingston, J.: *A Tiling Algorithm for High School Timetabling* in Burke, E., and Trick, M. (Eds.): *The Practice and Theory of Automated Timetabling V*. Springer, 2005.
11. Mamede, N., Rente, T.: *Repairing Timetables using Genetic Algorithms and Simulated Annealing* in Burke, E. and Carter, M. (Eds.): *The Practice and Theory of Automated Timetabling II*. Springer, 1997.
12. Michalewicz, Z.: *Genetic Algorithms + Data Structures = Evolution Programs, Second Edition*. Springer, 1994.
13. Mitchell, M.: *An Introduction to Genetic Algorithms*. MIT Press, 1998.
14. Ozcan, E., Alkan, A.: *Timetabling using a Steady State Genetic Algorithm* in Burke, E., Causmaecker, P. (Eds.): *The Practice and Theory of Automated Timetabling IV*. Springer, 2002.
15. Petrovski, A., Wilson, A., McCall, J.: *Statistical Analysis of Genetic Algorithms and Inference About Optimal Parameters*. Proceedings of Fifth Joint Conference on Information Sciences (JCIS 2000).
16. Rich, D.: *A Smart Genetic Algorithm for University Timetabling* in Burke, E., Ross. P. (Eds.): *The Practice and Theory of Automated Timetabling I*, pp. 181-197, Springer, 1995.

17. Ross, P, Hart, E, Corne, C.: *Some Observations about GA-Based Exam Timetabling*. In Burke, E. and Carter, M. (Eds.): *The Practice and Theory of Automated Timetabling II*, Pp. 115-129. Springer, 1997.
18. Tanaka, M., Adachi, S.: *Request-based Timetabling by Genetic Algorithm with Tabu Search*. Proceedings of the Fifth Joint Conference on Information Sciences (JCIS 2000) vol 1., pp 999-1002, Atlantic City, NJ, 2000. ISBN 0-9643456-9-2.
19. White, G., Zhang, J.: *Generating Complete University Timetables by Combining Tabu Search and Constraint Logic*. *The Practice and Theory of Automated Timetabling II*. Springer, 1997.

8.2. World Wide Web URLs

Dates last accessed are italicised in brackets

20. www.wikipedia.org – Free online encyclopaedia; used for introductory definitions of different algorithms (27/04/2005)
21. mathworld.wolfram.com/SimulatedAnnealing.html – a good summary of simulated annealing; part of a larger mathematics-oriented site (28/04/2005)
22. www.celcat.com – Homepage for the Celcat timetabling software (26/04/2005)
23. www.minitab.com – Homepage for the Minitab statistics package (26/04/2005)
24. www.spss.com – Homepage for the SPSS statistics package (26/04/2005)

8.3. Presentation

25. Paechter, B (Napier University): Presentation made at The Robert Gordon University, Aberdeen, November 1999.

Appendix A. Glossary

Algorithm A	One of the three timetabling algorithms being investigated. This is the simplest approach; a GA which assigns both times and rooms to modules. Each allele value in a chromosome represents a module; the value assigned to it is in the range of 0 – (number of timeslots X number of rooms)
Algorithm B	The second timetabling algorithm being investigated. A GA assigns a timeslot to each module. A greedy algorithm then assigns rooms to modules within each timeslot. Each allele again represents a single module; its range of values being equal to the number of timeslots.
Algorithm C	An MA based on the GA used in Algorithm A. Based on the literature survey and initial testing, this was unlikely to perform well and was abandoned early in the practical implementation in favour of developing the other three algorithms.
Algorithm D	The final timetabling algorithm covered. A duplicate of Algorithm B but using an MA instead of a GA.
Allele	A single value in a GA chromosome. In the timetabling algorithms, each allele is the timeslot (and room in one case) allocated to one particular module.
Chromosome	The group of allele values representing a single possible solution to the problem being solved by a GA.
Cohort	A group of students studying the same syllabus. This group may share modules with other cohorts.
Crossover	The process within a GA by which parts of multiple chromosomes (parents) are combined to generate one or more new chromosomes (offspring).
Elitism	The preservation of a number of the best chromosomes from one generation to the next in a GA.
Fitness	A numeric value calculated to represent how good a solution a given chromosome is.
GA or Genetic Algorithm	An algorithm based on the mechanics of Darwinian evolution. Possible solutions to a problem are encoded into strings of values called chromosomes which are given a “fitness” value derived from how good a solution they are. Some of these (with a bias toward fitter ones) are then combined with each other to produce new chromosomes or “offspring”; an element of random mutation is also present. This process is repeated either a fixed number of times or until a chromosome of a particular fitness is reached.

Generation	The population of chromosomes in a GA at one point in time. Each generation, a new population is created from a selection of the chromosomes in the previous one.
Greedy Algorithm	An algorithm which works towards a goal in stages. Used in Timetabling Algorithms B and D to take each room in turn from a given timeslot and allocate a room to it.
Hillclimbing	A local search algorithm which finds the nearest local optimum to a given solution. It slightly mutates the current solution and keeps the best solution found so far. This is repeated until the point is reached where no further improvements are possible without large changes to the current optimal solution.
Lecturer	The individual supervising or teaching a particular module. Multiple lecturers may also share modules.
Local optimum	Within a search space of solutions to a problem there is often a number of solutions which are higher in fitness than any similar solutions but are not the best overall.
Local Search	A term for the process of exploring the search space near to a given solution by altering it in small steps.
MA or Memetic Algorithm	An extension of a GA with an element of local search added. Every time a new chromosome is created a hillclimbing algorithm is used to mutate it to a local optimum.
Module	In the context of this project a module represents a single interaction between lecturer and group of students; a single lecture, tutorial or lab session.
Mutation	A random alteration of part or all of a newly created chromosome in a GA.
Node	A node in the timetable is a reference to a particular module requiring scheduling, which also stores the timeslot given to the module, the room assigned to it and links neighbouring modules (that is, ones it may not share a timeslot with)
NP-hard	Nondeterministic Polynomial - Hard; a class of problems which have no known algorithm that can solve them in polynomial time.
Recombination	See <i>Crossover</i>
Selection	The process of choosing chromosomes from a GA's current population which will contribute to the creation of the next generation.

Appendix B. Pseudocode

Rather than include the rather verbose Java source code in printed form, the important algorithms are outlined here in pseudocode. The Java source is included on the accompanying CD-ROM.

B.1. Local search

1. `countdown` = `LocalSearchIterations` parameter
2. `bestfitness` = current chromosome's fitness
3. `bestchromosome` = current chromosome
4. While `countdown` > 0, repeat:
 - 4.1. `change`=0
 - 4.2. `newChromosome` = `bestChromosome`
 - 4.2. For each clashing node in `newChromosome`:
 - 4.2.1. Repeat until no clashes found (max 20 times):
 - 4.2.1.1. Assign a random timeslot to node
 - 4.3. If `newChromosome's fitness` > `bestFitness`
 - 4.3.1. `bestfitness` = `newChromosome's fitness`
 - 4.3.2. `bestchromosome` = `newChromosome`
 - 4.4. Otherwise
 - 4.5. Subtract one from `countdown`

B.2. Local fitness

Incomplete; soft constraints calculation, and final total function. Parameters are the node being examined and its new and old timeslots

1. Set all clash counts to 0
2. Run greedy room allocator on modules in new and oldtimeslots
3. For each node visited by the greedy room allocator, repeat:
 - 3.1. If node has no room allocated, add one to room unallocated count
 - 3.2. Add the number of clashes the current node has with neighbours to clash count
 - 3.3. If node results in a module being split over a day boundary, add one to broken classes count
4. For each cohort taking the node, repeat:
 - 4.1. For each day on the timetable, repeat:
 - 4.1.1. If there is no unoccupied lunch hour, add one to lunch hour count
 - 4.1.2. If this is a Wednesday and there is a class in the afternoon, add one to Wednesday free count
 - 4.1.3. If there is a long run of classes, add one to class groupings count
 - 4.1.4. If there is a large gap between classes, add one to class spacings count
5. For each lecturer associated with the node, repeat:
 - 5.1. Set `dayFree` to false
 - 5.2. For each day on the timetable, repeat:
 - 5.2.1. If there is no unoccupied lunch hour, add one to lunch hour count
 - 5.2.2. If there are no classes today, set `dayFree` to true

- 5.3. If `dayFree` is false, add one to day free count
6. Add all the counts together, multiplying each by its weighting first
7. Return $1 / (\text{total} + 1)$

Part (7) above is one of the factors contributing to the incorrect operation of the local fitness function. However, during debugging the total number of constraint violations was not correct either. These problems would have to be solved in any further work on the memetic algorithm.

B.3. Fitness

1. Set all clash counts to 0
2. Reset the clashing nodes array
3. For each node (module), repeat:
 - 3.1. If node has no timeslot, add one to timeslot count
 - 3.2. If node has no room, add one to room count
 - Otherwise:
 - 3.2.1. If room size is exceeded, add one to room size count
 - 3.2.2. If room type is incorrect, add one to room type count
 - 3.3. Add number of neighbours with same timeslot (clashes) to clashes count
 - 3.4. If node results in a module being split over a day boundary, add one to broken classes count
4. For each cohort, repeat:
 - 4.1. For each day on the timetable, repeat:
 - 4.1.1. If there is no unoccupied lunch hour, add one to lunch hour count
 - 4.1.2. If this is a Wednesday and there is a class in the afternoon, add one to Wednesday free count
 - 4.1.3. If there is a long run of classes, add one to class groupings count
 - 4.1.4. If there is a large gap between classes, add one to class spacings count
5. For each lecturer, repeat:
 - 5.1. Set `dayFree` to false
 - 5.2. For each day on the timetable, repeat:
 - 5.2.1. If there is no unoccupied lunch hour, add one to lunch hour count
 - 5.2.2. If there are no classes today, set `dayFree` to true
 - 5.3. If `dayFree` is false, add one to day free count
6. Add all the counts together, multiplying each by its weighting first
7. Return $1 / (\text{total} + 1)$

B.4. Number of Clashes

1. Set `rval` to 0
2. For each neighbour of the current node on the timetable graph, repeat:
 - 2.1. If the index of the neighbour is greater than that of current node:
 - 2.1.1. If the neighbour has a timeslot assigned to it:
 - 2.1.1.1. Set `thisStart` to the timeslot of the current node
 - 2.1.1.2. Set `thisEnd` to `thisStart` + current node's length

- 2.1.1.3. Set `neighbourStart` to timeslot of neighbour
- 2.1.1.4. Set `neighbourEnd` to `neighbourStart + neighbour's length`
- 2.1.1.5. If both `(thisEnd < neighbourStart)` and `(thisStart > neighbourEnd)` are false, add one to `rval`

3. Return `rval`

B.5. Greedy room allocator

This assumes nodes and rooms are already sorted in size order.

1. Get the set of timetables allocated to the specified timeslot
2. Get set of rooms available at this timeslot from the Timeslots object
3. Set `curRoom` to index of last room in array
4. Set `curMod` to index of last node (module) in list
5. While both `curRoom` and `curMod` are greater than or equal to zero, repeat:
 - 5.1. Set status of current node to visited (used by local fitness / search)
 - 5.2. If the room at index `curRoom` is big enough to hold the module at index `curMod`:
 - 5.2.1. Assign the room `curRoom` to the module `curMod`
 - 5.2.2. Decrement `curMod` by one (move on to next module)
 - 5.2.3. Decrement `curRoom` by one (move on to next module)
 - 5.3. Otherwise:
 - 5.3.1. Decrement `curRoom` by one (try next room)
6. Return the list of nodes visited

B.6. Copy alleles to timetable

The process of copying the allele values from a chromosome to the Timetable object for evaluation has two different versions; one for Algorithm A and one for B and D.

Algorithm A:

1. For each allele value (and corresponding module in the timetable), repeat:
 - 1.1. Set `timeslot` to the allele value mod (the number of timeslots)
 - 1.2. Set `room` to the allele value divided by the number of timeslots
 - 1.3. Save room and timeslot to appropriate node in timetable

Algorithm B & D:

1. For each allele value (and corresponding module in the timetable), repeat:
2. Run greedy room allocator for each timeslot

B.7. Crossover

Performed when generating a new population. Rather than always just copying parents to make children, on random occasions (with a given probability) run the crossover algorithm on the child chromosomes.

1. Create an array of Booleans the same length as the chromosome
2. Repeat a number of times equal to the required number of crossover points:
 - 2.1. Pick a random point in the array

- 2.2. Set it to true
3. Set `swap` to false
3. For each allele value in the chromosome, repeat:
 - 3.1. If `swap` is true, copy alleles into corresponding child chromosome alleles
 - 3.2. Otherwise, copy alleles into opposite child chromosome alleles
 - 3.3. If the Boolean array value for this allele is true, flip the value of `swap`

B.8. Number Creep Mutation

Performed when generating a new population. Rather than always just copying parents to make children, on random occasions (with a given probability) run the mutation algorithm on the child chromosomes. Random number generation is similar but the allele value is just replaced with a value between 0 and `rangemax`. An argument passed to the `mutate` method denotes the probability that a single allele will mutate (in this project, this is always 0.1) – this is different to the probability of a mutation occurring when generating a new chromosome.

1. For each allele in the chromosome, repeat:
 - 1.1. Pick a random value between 0 and 1.
 - 1.2. Multiply this value by the maximum creep step time 2.
 - 1.3. Subtract the maximum creep step size from this
 - 1.4. Add this value to the allele
 - 1.5. If the allele is out of range, set it to the limit it is closest to

B.9. Tournament Selection

1. Repeat for the number of chromosomes that need to be selected (usually equal to the size of the population):
 - 1.1. Pick two chromosomes at random from the population
 - 1.2. Pick a random number between 0 and 1
 - 1.3. If the number is under 0.8, add the fitter chromosome to those selected
 - 1.4. Otherwise, add the less fit chromosome to those selected
2. Return the set of selected chromosomes

B.10. Boltzmann Selection

Boltzmann selection is very similar to the more widely described roulette wheel selection. In the `Population` class, both are implemented in the same method, with a single parameter to switch between them. An extra parameter sets the selection pressure.

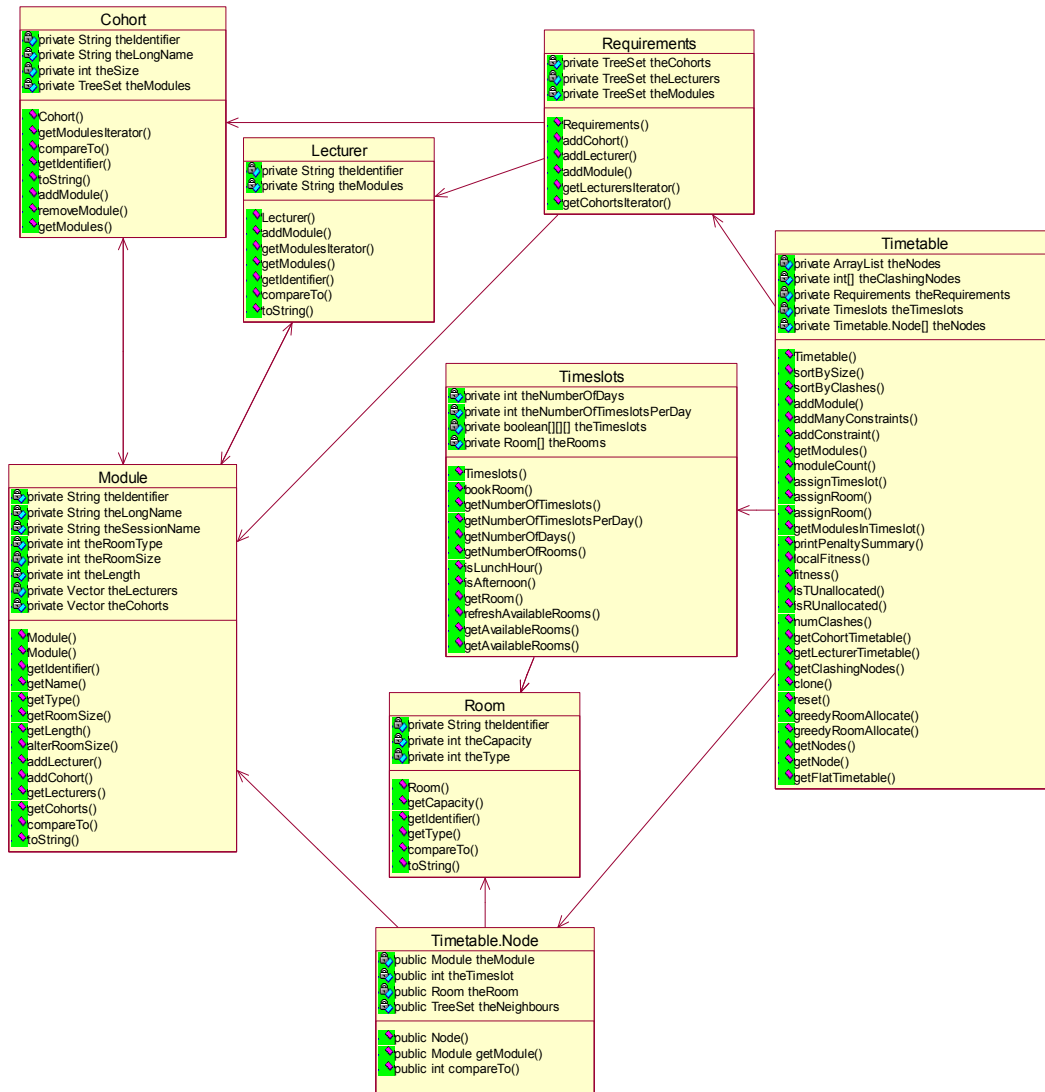
1. Initialise an array equal in size to the population: `cumulativeFitnesses`
2. Set `totalFitness` to 0
3. For each chromosome in the population, repeat:
 - 3.1. Divide the average fitness of the population by the pressure
 - 3.2. Divide the fitness of the current chromosome by the pressure
 - 3.3. Divide the value found in 3.2 by the value found in 3.1.
 - 3.4. Add the value from 3.3 to `totalFitness`
 - 3.5. Set the value at the current position in the `cumulativeFitnesses` array to `totalFitness`.
4. Repeat for the number of chromosomes that need to be selected (usually equal to the size of the population):

- 4.1. Pick a random number between 0 and 1
- 4.2. For each element in the `cumulativeFitnesses` array, repeat:
 - 4.2.1. Divide the value from the array by `totalFitness`
 - 4.2.2. If this value is greater than the random number chosen, or `totalFitness` is 0 then add the chromosome from the position corresponding with the current position in the `cumulativeFitnesses` array to the set of selected chromosomes.

Appendix C. Class Diagrams

For simplicity all method parameters, helper methods, debugging methods (such as toString) and any static final variables have been omitted.

C.1. Timetable Classes



C.2. GA Classes



Appendix D. Sample Timetables

Some example timetables viewed in the debugging GUI. Note that in the last couple there are “nameless” classes – these are artefacts from the source timetable data. All timetables have an hour within the lunch period, lectures/tutorials have been correctly assigned to the right room type, as have labs. Wednesday afternoons have not been avoided in all cases – this could be rectified by setting a higher finishing fitness (so more soft constraints have to be met) or by adjusting the weights of the constraints.

Timetable Application								
Lecturer: Ahriz H								
	0	1	2	3	4	5	6	7
Day 1		CM1009, T, B39				CM1005, T, C39		
Day 2	CM1009, LAB, C27/C28		CM1003, LAB, C8/C8A	CM1005, LAB, C8/C8A			CM1004, LAB, C8/C8A	
Day 3		CM1009, LAB, C8/C8A	, T, B39					
Day 4								
Day 5	CM1003, T, A23			CM1009, T, C40ZCL				CM1003, T, C47

Timetable Application								
Lecturer: Webster M								
	0	1	2	3	4	5	6	7
Day 1							CM1003, LAB, C27/C28	
Day 2								
Day 3								
Day 4			CM1900, L, A23			CM1900, L, C39		
Day 5		CM1900, T, B39	CM1003, T, A23					

Timetable Application								
Cohort: Foundation Course Group A,28								
	0	1	2	3	4	5	6	7
Day 1	CM1005, L, C47		CM1003, LAB, C8/C8A					
Day 2	CM1004, L, C47						CM1004, LAB, C8/C8A	
Day 3						CM1010, LAB, C8/C8A	CM1010, L, C47	
Day 4						CM1005, LAB, C8/C8A		
Day 5	CM1005, T, C47	CM1003, L, C47	, T, C47	CM1010, T, C47				

Timetable Application								
Cohort: Foundation Course Group C,27								
	0	1	2	3	4	5	6	7
Day 1	CM1005, L, C47			CM1005, T, B39			CM1003, LAB, C27/C28	
Day 2	CM1004, L, C47		CM1010, T, B39					
Day 3					CM1010, LAB, C8/C8A		CM1010, L, C47	
Day 4						CM1004, LAB, C27/C28		
Day 5		CM1003, L, C47	CM1003, T, A23	, T, B39				

Appendix E. Data from Experiments

E.1. Fractional Factorial – Algorithms A and B

Population Size	Mutation Rate	Crossover Rate	Crossover Points	Mutation Creep Step	Selection Method	Best fitness found by A in 200 gens	Best fitness found by B in 200 gens
100	0.02	0.25	20	1	Tournament	0.002043	0.001646
100	0.02	0.75	20	1	Tournament	0.0023	0.001754
500	0.02	0.75	2	10	Tournament	0.00264	0.001892
100	0.2	0.75	20	1	Boltzmann	0.001892	0.001477
100	0.2	0.25	2	1	Tournament	0.002115	0.001682
500	0.2	0.75	2	1	Boltzmann	0.001975	0.001505
500	0.2	0.75	20	10	Tournament	0.002507	0.001539
500	0.02	0.75	20	10	Tournament	0.002451	0.001744
100	0.02	0.75	2	1	Boltzmann	0.002022	0.001517
100	0.2	0.25	2	10	Boltzmann	0.00194	0.00151
100	0.02	0.25	20	10	Tournament	0.002077	0.001654
500	0.02	0.25	2	10	Boltzmann	0.002046	0.001623
500	0.2	0.75	20	1	Boltzmann	0.001949	0.001482
500	0.2	0.25	2	1	Boltzmann	0.00203	0.001535
500	0.02	0.25	2	1	Boltzmann	0.00204	0.00158
500	0.02	0.25	2	1	Tournament	0.002277	0.00173
500	0.02	0.75	2	10	Boltzmann	0.001944	0.001535
100	0.02	0.25	2	10	Tournament	0.002015	0.001587
100	0.2	0.75	2	10	Boltzmann	0.001949	0.001497
500	0.02	0.75	20	1	Tournament	0.002495	0.001784
100	0.2	0.75	20	10	Boltzmann	0.001885	0.001478
500	0.2	0.75	2	10	Tournament	0.002365	0.001627
100	0.02	0.25	20	10	Boltzmann	0.001968	0.001572
500	0.2	0.25	2	10	Tournament	0.002281	0.001792
100	0.02	0.25	20	1	Boltzmann	0.002015	0.001569
100	0.02	0.25	2	10	Boltzmann	0.001996	0.00158
100	0.2	0.25	2	10	Tournament	0.002111	0.001658
100	0.2	0.25	2	1	Boltzmann	0.001966	0.001506
100	0.02	0.75	20	10	Tournament	0.002305	0.001774
500	0.02	0.75	2	1	Boltzmann	0.001976	0.001527
500	0.2	0.75	20	1	Tournament	0.002495	0.001784
500	0.02	0.75	20	10	Boltzmann	0.001915	0.00151

E.2. Fractional Factorial – Algorithm D

Population size	Mutation Rate	Crossover Rate	Crossover Points	Mutation Creep Step	Selection Method	Local Search Iterations	Best fitness found by D in 200 gens
100	0.2	0.25	2	1	Tournament	10	0.001666
100	0.2	0.25	20	10	Boltzmann	2	0.001509
100	0.02	0.75	2	1	Boltzmann	2	0.001661
500	0.02	0.25	20	1	Tournament	2	0.001821
500	0.02	0.75	20	1	Boltzmann	2	0.001516

Population size	Mutation Rate	Crossover Rate	Crossover Points	Mutation Creep Step	Selection Method	Local Search Iterations	Best fitness found by D in 200 gens
500	0.02	0.25	2	10	Tournament	2	0.001747
100	0.2	0.25	20	1	Boltzmann	10	0.001499
100	0.2	0.75	20	1	Boltzmann	2	0.001476
500	0.02	0.25	20	1	Boltzmann	10	0.001512
100	0.02	0.75	2	10	Boltzmann	10	0.001569
100	0.02	0.25	20	1	Boltzmann	2	0.001569
500	0.2	0.25	2	10	Boltzmann	2	0.001507
100	0.02	0.75	2	10	Tournament	2	0.001645
500	0.2	0.75	2	1	Boltzmann	2	0.001476
100	0.2	0.25	20	10	Tournament	10	0.00163
100	0.2	0.75	20	10	Boltzmann	10	0.001469
100	0.02	0.75	20	1	Tournament	2	0.001773
100	0.2	0.75	2	10	Boltzmann	2	0.001512
500	0.2	0.25	20	1	Boltzmann	2	0.001503
500	0.2	0.75	20	1	Boltzmann	10	0.001478
100	0.02	0.25	2	10	Tournament	10	0.001598
500	0.02	0.75	2	10	Boltzmann	2	0.001537
500	0.02	0.75	2	10	Tournament	10	0.001857
500	0.2	0.75	2	1	Tournament	10	0.001733
100	0.2	0.25	2	1	Boltzmann	2	0.001516
500	0.2	0.25	20	10	Boltzmann	10	0.001482
100	0.02	0.25	20	1	Tournament	10	0.00164
500	0.02	0.25	20	10	Tournament	10	0.001811
500	0.02	0.75	20	10	Boltzmann	10	0.001507
500	0.02	0.75	20	1	Tournament	10	0.001809
500	0.02	0.25	2	10	Boltzmann	10	0.001588
100	0.02	0.2	20	1	Boltzmann	10	0.001493

E.3. Response Surface – Algorithms A and B

Population size	Mutation Rate	Crossover Rate	Crossover Points	Mutation Creep Step	Best fitness found by A in 200 gens	Best fitness found by B in 200 gens
500	0.11	0.5	11	6	0.002302	0.001623
300	0.11	0.5	29	6	0.002393	0.001691
300	0.11	0.5	11	6	0.002305	0.001696
500	0.2	0.25	2	10	0.002278	0.001787
100	0.02	0.75	20	10	0.002278	0.001753
300	0.29	0.5	11	6	0.002137	0.001559
500	0.02	0.75	20	1	0.002764	0.0018
500	0.02	0.25	2	1	0.002199	0.001719
300	0.11	0.5	11	6	0.002305	0.001696
300	0.11	0.5	1	6	0.002213	0.001675
300	0.11	1	11	6	0.002168	0.001562
300	0.11	0.5	11	6	0.002305	0.001696
100	0.2	0.75	20	1	0.002215	0.00161
300	0.11	0.5	11	6	0.002305	0.001696
300	0.11	0.5	11	6	0.002305	0.001696
500	0.02	0.25	20	10	0.002318	0.00182
500	0.2	0.25	20	1	0.002302	0.001777

Population size	Mutation Rate	Crossover Rate	Crossover Points	Mutation Creep Step	Best fitness found by A in 200 gens	Best fitness found by B in 200 gens
300	0.11	0	11	6	0.002032	0.001635
100	0.2	0.25	2	1	0.002104	0.001696
300	0.11	0.5	11	1	0.002291	0.00152
100	0.2	0.75	2	10	0.002064	0.001629
500	0.2	0.75	20	10	0.002395	0.001549
500	0.02	0.75	2	10	0.002453	0.001904
300	0.11	0.5	11	6	0.002305	0.001696
100	0.02	0.25	2	10	0.00202	0.001583
100	0.11	0.5	11	6	0.002157	0.001753
300	0.01	0.5	11	6	0.00231	0.001818
100	0.02	0.25	20	1	0.002059	0.001636
500	0.2	0.75	2	1	0.002451	0.001747
100	0.2	0.25	20	10	0.002098	0.001657
300	0.11	0.5	11	10	0.002285	0.00171
100	0.02	0.75	2	1	0.002191	0.001714

E.4. Response Surface – Algorithm D

Population size	Mutation Rate	Crossover Rate	Crossover Points	Mutation Creep Step	Local Search Iterations	Best fitness found by D in 200 gens
300	0.2	0.5	11	6	6	0.001567
384	0.14784	0.605112	15	4	4	0.001642
300	0.11	0.5	11	6	10	0.00172
300	0.11	0.5	11	6	6	0.001635
300	0.11	0.5	11	6	6	0.001635
384	0.07216	0.394888	15	4	4	0.001837
300	0.11	0.5	11	10	6	0.001631
384	0.14784	0.394888	15	6	7	0.001764
384	0.14784	0.605112	15	7	7	0.001459
384	0.14784	0.605112	7	4	7	0.001508
300	0.11	0.5	11	6	1	0.0015
300	0.11	0.5	11	6	6	0.001527
216	0.07216	0.394888	15	4	7	0.001711
216	0.07216	0.605112	15	4	4	0.001623
300	0.11	0.5	2	6	6	0.00169
216	0.14784	0.605112	7	7	7	0.001469
384	0.07216	0.605112	7	7	7	0.001561
216	0.07216	0.605112	7	7	4	0.001577
216	0.07216	0.605112	15	7	7	0.001619
216	0.14784	0.394888	15	4	4	0.00157
300	0.11	0.5	11	6	6	0.001527
300	0.11	0.5	11	1	6	0.001622
384	0.07216	0.605112	15	7	4	0.00151
216	0.14784	0.605112	15	4	7	0.001507
216	0.14784	0.605112	15	7	4	0.001481
384	0.07216	0.605112	7	4	4	0.0016
300	0.11	0.5	20	6	6	0.001591
384	0.07216	0.394888	7	4	7	0.001669
216	0.14784	0.394888	15	7	7	0.001631

Population size	Mutation Rate	Crossover Rate	Crossover Points	Mutation Creep Step	Local Search Iterations	Best fitness found by D in 200 gens
300	0.02	0.5	11	6	6	0.001663
216	0.14784	0.394888	7	4	7	0.001574
216	0.07216	0.394888	7	4	4	0.001605
100	0.11	0.5	11	6	6	0.001566
300	0.11	0.75	11	6	6	0.001477
384	0.14784	0.605112	7	7	4	0.001481
384	0.07216	0.394888	15	7	7	0.001641
384	0.07216	0.605112	15	4	7	0.001563
384	0.14784	0.394888	15	7	4	0.001531
216	0.14784	0.394888	7	7	4	0.001712
300	0.11	0.5	11	6	6	0.001635
216	0.07216	0.394888	7	7	7	0.00179
216	0.07216	0.605112	7	4	7	0.00176
300	0.11	0.5	11	6	6	0.001614
300	0.11	0.5	11	6	6	0.00152
384	0.14784	0.394888	7	7	7	0.001489
300	0.11	0.5	11	6	6	0.00152
216	0.07216	0.394888	15	7	4	0.001694
500	0.11	0.5	11	6	6	0.00152
300	0.11	0.5	11	6	6	0.00152
216	0.14784	0.605112	7	4	4	0.001476
384	0.14784	0.394888	7	4	4	0.001497
384	0.07216	0.394888	7	7	4	0.001699
300	0.11	0.25	11	6	6	0.001607

E.5. Confirmation Experiment

Run 10 times, these results were produced.

Best fitness found by A in 200 gens	Best fitness found by B in 200 gens	Best fitness found by D in 200 gens
0.002549	0.001694	0.001612
0.002432	0.001691	0.001587
0.002467	0.001732	0.001464
0.002492	0.001597	0.001633
0.002437	0.001714	0.001592
0.002530	0.001610	0.001599
0.002426	0.001712	0.001652
0.002408	0.001720	0.001594
0.002530	0.001688	0.001631
0.002420	0.001720	0.001597
0.002486	0.001671	0.001655

E.6. Comparison Experiments

Each of the comparison experiments (repeated 10 times for each algorithm) ran to several thousand generations. This is too large a quantity of data to

realistically reproduce here; the output from the experiments may be found in the output folder on the accompanying CD.